Q1. Write a program to find all pairs of an integer array whose sum is equal to a given number?

In [8]:
```
from itertools import combinations

def findPairs(lst, k):

    return [pair for pair in combinations(lst, 2) if sum(pair) == k]


lst=list(map(int,input("Please Enter integer element of an array separated by comma").split(",")))
k = int(input("Please enter the sum value"))
print(findPairs(lst, k))
```

```
Please Enter integer element of an array separated by comma5,7,11,32
Please enter the sum value16
[(7, 11)]
```

Q2. Write a program to reverse an array in place? In place means you cannot create a new array. You have to update the original array

In [13]:
```
def ReverseArray(A):
    print( A[::-1])

A = list(map(int,input("Enter element of an array separated by comma").split(",")))
print("Original array is")
print(A)
print("Reversed array is")
ReverseArray(A)
```

```
Enter element of an array separated by comma1,2,3,4,5,6
Original array is
[1, 2, 3, 4, 5, 6]
Reversed array is
[6, 5, 4, 3, 2, 1]
```

Q3. Write a program to check if two strings are a rotation of each other?

In [16]:
```
def checkRotation(s1, s2):
    temp = ''

    if len(s1) != len(s2):
        return False
    temp = s1 + s2

    if s2 in temp:
        return True
    else:
        return False

string1 = "PYTHON"
string2 = "THONPY"

if checkRotation(string1, string2):
    print("Given Strings are rotation of each other")
else:
    print("Given Strings are not rotation of each other")
```

```
Given Strings are rotation of each other
```

Q4. Write a program to print the first non-repeated character from a string?

In [17]:
```
from collections import Counter

def printNonrepeated(string):

    freq = Counter(string)

    for i in string:
        if(freq[i] == 1):
            print(i)
            break

string = "Linear Data Structures"

printNonrepeated(string)
```

```
L
```

Q5. Read about the Tower of Hanoi algorithm. Write a program to implement it.

In [23]:
```
#Creating a recursive function
def Tower_of_hanoi(disks,source,auxiliary,target):
    if(disks==1):
        print('Move disk 1 from rod {} to rod {}'.format(source,target))
        return

#Function call itself
    Tower_of_hanoi(disks - 1,source,target,auxiliary)
    print('Move disk {} from rod {} to rod {}'.format(disks,source,target))
    Tower_of_hanoi(disks - 1,auxiliary,source,target)

disks = int(input('Enter the number of disks: '))

#Calling the function
Tower_of_hanoi(disks,'P','Q','R')
```

```
Enter the number of disks: 3
Move disk 1 from rod P to rod R
Move disk 2 from rod P to rod Q
Move disk 1 from rod R to rod Q
Move disk 3 from rod P to rod R
Move disk 1 from rod Q to rod P
Move disk 2 from rod Q to rod R
Move disk 1 from rod P to rod R
```

Q6. Read about infix, prefix, and postfix expressions. Write a program to convert postfix to prefix expression.

In [24]:
```
def isOperator(x):

    if x == "+":
        return True

    if x == "-":
        return True

    if x == "/":
        return True

    if x == "*":
        return True

    return False

# Convert postfix to Prefix expression


def postToPre(post_exp):

    s = []

    # length of expression
    length = len(post_exp)

    # reading from right to left
    for i in range(length):

        # check if symbol is operator
        if (isOperator(post_exp[i])):

            # pop two operands from stack
            op1 = s[-1]
            s.pop()
            op2 = s[-1]
            s.pop()

            # concat the operands and operator
            temp = post_exp[i] + op2 + op1

            # Push string temp back to stack
            s.append(temp)

        # if symbol is an operand
        else:

            # push the operand to the stack
            s.append(post_exp[i])

    ans = ""
    for i in s:
        ans += i
    return ans


# Driver Code
if __name__ == "__main__":

    post_exp = "ab+cd-"

    # Function call
    print("Prefix : ", postToPre(post_exp))
```

```
Prefix :  +ab-cd
```

Q7. Write a program to convert prefix expression to infix expression.

In [25]:
```
def prefixToInfix(prefix):
    stack = []

    # read prefix in reverse order
    i = len(prefix) - 1
    while i >= 0:
        if not isOperator(prefix[i]):

            # symbol is operand
            stack.append(prefix[i])
            i -= 1
        else:

            # symbol is operator
            str = "(" + stack.pop() + prefix[i] + stack.pop() + ")"
            stack.append(str)
            i -= 1

    return stack.pop()

def isOperator(c):
    if c == "*" or c == "+" or c == "-" or c == "/" or c == "^" or c == "(" or c == ")":
        return True
    else:
        return False

# Driver Code
if __name__ == "__main__":
    str = "*-A/B/C-/AKL/"
    print(prefixToInfix(str))
```

```
((A-(B/(C/(A/K/L))))*(A-L))
```

Q8. Write a program to check if all the brackets are closed in a given code snippet.

In [26]:
```
def arebracketsbalanced(expr):
    stack = []

    # Traversing the Expression
    for char in expr:
        if char in ["(", "{", "["]:

            # Push the element in the stack
            stack.append(char)
        else:

            # If current character is not opening
            # bracket, then it must be closing.
            # So stack cannot be empty at this point.
            if not stack:
                return False
            current_char = stack.pop()
            if current_char == '(':
                if char != ")":
                    return False
            if current_char == '{':
                if char != "}":
                    return False
            if current_char == '[':
                if char != "]":
                    return False

    # Check Empty Stack
    if stack:
        return False
    return True

# Driver Code
if __name__ == "__main__":
    expr = "{()}[]"

    # Function call
    if arebracketsbalanced(expr):
        print("Balanced")
    else:
        print("Not Balanced")
```

```
Balanced
```

Q9. Write a program to reverse a stack.

In [31]:
```
class Stack:

    # create empty list
    def __init__(self):
        self.elements = []

    # push() for insert an element
    def push(self, value):
        self.elements.append(value)

    # pop() for remove an element
    def pop(self):
        return self.elements.pop()

    # empty() check the stack is empty of not
    def empty(self):
        return self.elements == []

    # show() display stack
    def show_stack(self):
        for value in reversed(self.elements):
            print(value)

# Insert_Bottom() insert value at bottom
def BottomInsert(s, value):

    # check the stack is empty or not
    if s.empty():

        # if stack is empty then call
        # push() method.
        s.push(value)

    # if stack is not empty then execute
    # else block
    else:
        popped = s.pop()
        BottomInsert(s, value)
        s.push(popped)

# Reverse() reverse the stack
def Reverse(s):
    if s.empty():
        pass
    else:
        popped = s.pop()
        Reverse(s)
        BottomInsert(s, popped)


# create object of stack class
stk = Stack()

stk.push(1)
stk.push(2)
stk.push(3)
stk.push(4)
stk.push(5)

print("Original Stack")
stk.show_stack()

print("\nStack after Reversing")
Reverse(stk)
stk.show()
```

```
Original Stack
5
4
3
2
1

Stack after Reversing
1
2
3
4
5
```

Q10. Write a program to find the smallest number using a stack.

In [33]:
```
class Node:
    # Node constructor which assign argument to node's value
    def __init__(self, value):
        self.value = value
        self.next = None

    # This method returns the string representation of the object.
    def __str__(self):
        return "Node({})".format(self.value)

    # __repr__ is same as __str__
    __repr__ = __str__


class Stack:
    # Stack constructor initialise top of stack and counter.
    def __init__(self):
        self.top = None
        self.count = 0
        self.minimum = None

    # This method returns the string representation of the object (stack).
    def __str__(self):
        temp = self.top
        out = []
        while temp:
            out.append(str(temp.value))
            temp = temp.next
        out = '\n'.join(out)
        return ('Top {}\nStack :\n{}'.format(self.top.out))

    # __repr__ is same as __str__
    __repr__ = __str__

    # This method is used to get minimum element of stack
    def getMinimum(self):
        if self.top is None:
            return "Stack is empty"
        else:
            print("Minimum Element in the stack is: {}".format(self.minimum))

    # Method to check if Stack is Empty or not
    def isEmpty(self):
        # If top equals to None then stack is empty
        if self.top == None:
            return True
        else:
            # If top not equal to None then stack is empty
            return False

    # This method returns length of stack
    def __len__(self):
        self.count = 0
        tempNode = self.top
        while tempNode:
            tempNode = tempNode.next
            self.count+=1
        return self.count

    # This method returns top of stack
    def getTop(self):
        if self.top is None:
            print ("Stack is empty")
        else:
            if self.top.value < self.minimum:
                print("Top Most Element is: {}".format(self.minimum))
            else:
                print("Top Most Element is: {}".format(self.top.value))

    # This method is used to add node to stack
    def push(self,value):
        if self.top == None:
            self.top = Node(value)
            self.minimum = value

        elif value < self.minimum:
            temp = (2 * value) - self.minimum
            new_node = Node(temp)
            new_node.next = self.top
            self.top = new_node
            self.minimum = value
        else:
            new_node = Node(value)
            new_node.next = self.top
            self.top = new_node
        print("Number Inserted: {}".format(value))

    # This method is used to pop top of stack
    def pop(self):
        if self.top is None:
            print( "Stack is empty")
        else:
            removedNode = self.top.value
            self.top = self.top.next
            if removedNode < self.minimum:
                print ("Top Most Element Removed :{} ".format(self.minimum))
                self.minimum = ( 2 * self.minimum ) - removedNode
            else:
                print ("Top Most Element Removed : {}".format(removedNode))

# Driver program to test above class
stack = Stack()

stack.push(3)
stack.push(5)
stack.getMin()
stack.push(2)
stack.push(1)
stack.getMin()
stack.pop()
stack.getMin()
stack.pop()
stack.peek()
```

```
Number Inserted: 3
Number Inserted: 5
Minimum Element in the stack is: 3
Number Inserted: 2
Number Inserted: 1
Minimum Element in the stack is: 1
Top Most Element Removed : 1
Minimum Element in the stack is: 2
Top Most Element Removed : 2
Top Most Element is: 5
```