Multiplication table

In [1]:

```
num = 12
for i in range(1,11):
    print(num,'x',i,'=',num*i)
```

```
12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
12 x 4 = 48
12 x 5 = 60
12 x 6 = 72
12 x 7 = 84
12 x 8 = 96
12 x 9 = 108
12 x 10 = 120
```

prime no

In [1]:

```
num=11
if num>1:
    for i in range(2,int(num/2)+1):
        if (num%i)==0:
            print(num,"is not a prime number")
            break

    else:
        print(num,"is a prime number")
else:
    print(num,"is not a prime number")
```

```
11 is a prime number
```

Factorial

In [5]:

```
num=int(input('enter the number'))
#num=23
fact=1
for i in range(1,num+1):
    fact=fact*i
print("The Factorial of 23 is:",fact)
```

```
enter the number23
The Factorial of 23 is: 25852016738884976640000
```

even or odd

In [8]:

```
num = 7
if num%2 ==0:
    print(num,"The number is even")
else:
    print(num,"The number is odd")
```

```
7 The number is odd
```

calculator

In [11]:

```
def add(x,y):
    return x+y
def sub(x,y):
    return x-y
def multiply(x,y):
    return x*y
def divide(x,y):
    return x/y
num1=12
num2=4
print(add(num1,num2),"Addition of num1 and num2")
print(sub(num1,num2),"substraction of num1 and num2")
print(multiply(num1,num2),"multiply of num1 and num2")
print(divide(num1,num2),"Divide of num1 and num2")
```

```
16 Addition of num1 and num2
8 substraction of num1 and num2
48 multiply of num1 and num2
3.0 Divide of num1 and num2
```

list operations

In [1]:

```python
# Creating a list
my_list = [1, 2, 3, 4, 5]
# Accessing elements
print("Original List:", my_list)
print("Element at index 2:", my_list[2])
# Slicing
print("Sliced List (index 1 to 3):", my_list[1:4])
# Modifying elements
my_list[2] = 6
print("Modified List:", my_list)
# Appending and extending
my_list.append(7)
print("List after appending 7:", my_list)
my_list.extend([8, 9])
print("List after extending with [8, 9]:", my_list)
# Removing elements
my_list.remove(6)
print("List after removing 6:", my_list)
popped_element = my_list.pop(2)
print(f"Popped element at index 2: {popped_element}, Updated List: {my_list}")
# Finding index of an element
index_of_5 = my_list.index(5)
print("Index of 5:", index_of_5)
# Length of the list
list_length = len(my_list)
print("Length of the list:", list_length)
# Check if an element is in the list
element_to_check = 8
if element_to_check in my_list:
 print(f"{element_to_check} is in the list.")
else:
 print(f"{element_to_check} is not in the list.")
```

```
Original List: [1, 2, 3, 4, 5]
Element at index 2: 3
Sliced List (index 1 to 3): [2, 3, 4]
Modified List: [1, 2, 6, 4, 5]
List after appending 7: [1, 2, 6, 4, 5, 7]
List after extending with [8, 9]: [1, 2, 6, 4, 5, 7, 8, 9]
List after removing 6: [1, 2, 4, 5, 7, 8, 9]
Popped element at index 2: 4, Updated List: [1, 2, 5, 7, 8, 9]
Index of 5: 2
Length of the list: 6
8 is in the list.
```

list Methods

In [3]:

```python
# Creating a list
my_list = [1, 2, 3, 4, 5]
# Method: append
my_list.append(6)
print("List after append(6):", my_list)
# Method: extend
my_list.extend([7, 8])
print("List after extend([7, 8]):", my_list)
# Method: insert
my_list.insert(2, 10)
print("List after insert(2, 10):", my_list)
# Method: remove
my_list.remove(4)
print("List after remove(4):", my_list)
# Method: pop
popped_element = my_list.pop(1)
print(f"Popped element at index 1: {popped_element}, Updated List: {my_list}")
# Method: index
index_of_3 = my_list.index(3)
print("Index of 3:", index_of_3)
# Method: count
count_of_6 = my_list.count(6)
print("Count of 6:", count_of_6)
# Method: reverse
my_list.reverse()
print("Reversed List:", my_list)
# Method: sort
my_list.sort()
print("Sorted List:", my_list)
# Method: copy
copied_list = my_list.copy()
print("Copied List:", copied_list)
# Method: clear
my_list.clear()
print("Cleared List:", my_list)
```

```
List after append(6): [1, 2, 3, 4, 5, 6]
List after extend([7, 8]): [1, 2, 3, 4, 5, 6, 7, 8]
List after insert(2, 10): [1, 2, 10, 3, 4, 5, 6, 7, 8]
List after remove(4): [1, 2, 10, 3, 5, 6, 7, 8]
Popped element at index 1: 2, Updated List: [1, 10, 3, 5, 6, 7, 8]
Index of 3: 2
Count of 6: 1
Reversed List: [8, 7, 6, 5, 3, 10, 1]
Sorted List: [1, 3, 5, 6, 7, 8, 10]
Copied List: [1, 3, 5, 6, 7, 8, 10]
Cleared List: []
```

In [ ]:

```python
import time
now = time.ctime()
def simple_chatbot(user_input):
 conversations = {
 "hi": "Hello! How can I help you?",
 "how are you": "I'm doing well, thank you. How about you?",
 "name": "I'm a chatbot. You can call me ChatPy!",
 "age": "I don't have an age. I'm just a program.",
 "bye": "Goodbye! Have a great day.",
 "python": "Python is a fantastic programming language!",
 "weather": "I'm sorry, I don't have real-time data. You can check a weather website for updates.",
 "help": "I'm here to assist you. Ask me anything!",
 "thanks": "You're welcome! If you have more questions, feel free to ask.",
 "default": "I'm not sure how to respond to that. You can ask me something else.",
  "what is the time now": now,
 }
 # Convert user input to lowercase for case-insensitive matching
 user_input_lower = user_input.lower()
 # Retrieve the response based on user input
 response = conversations.get(user_input_lower, conversations["default"])
 return response
# Chatbot interaction loop
print("Hello! I'm ChatPy, your friendly chatbot.")
print("You can start chatting. Type 'bye' to exit.")
while True:
 user_input = input("You: ")

 if user_input.lower() == 'bye':
     print("ChatPy: Goodbye! Have a great day.")
     break
 response = simple_chatbot(user_input)
 print("ChatPy:", response)
```

```
Hello! I'm ChatPy, your friendly chatbot.
You can start chatting. Type 'bye' to exit.
You: what is the time now
ChatPy: Thu Dec 28 14:50:11 2023
```

set operations

In [11]:

```
set1={1,2,3,4,5}
set2={3,4,5,6,7}
#union of set 1 and set2
union_set=set1.union(set2)
print(union_set,"union set")
#intersection of set1 and set2
intersection_set=set1.intersection(set2)
print(intersection_set,"intersection set")
#Difference of set1 and set2
difference_set1=set1.difference(set2)
print(difference_set1,"set1-set2")
#symmetric difference
symmetric_difference_set=set1.symmetric_difference(set2)
print(symmetric_difference_set,"symmetric difference set")
#check if sets have common elements
have_common_elements=set1.isdisjoint(set2)
print("Do set1 and set2 have any common elements?",not have_common_element
s)
#Adding an element to a set
set1.add(6)
print("set1 after adding element:",set1)
#Removing an element from a set
set1.remove(3)
print("set1 after removing element:",set1)
```

```
{1, 2, 3, 4, 5, 6, 7} union set
{3, 4, 5} intersection set
{1, 2} set1-set2
{1, 2, 6, 7} symmetric difference set
Do set1 and set2 have any common elements? True
set1 after adding element: {1, 2, 3, 4, 5, 6}
set1 after removing element: {1, 2, 4, 5, 6}
```

Printing even numbers from a list

In [1]:

```
l1=[1,2,3,4,5,6,7,8,9,10]
l2=[]
for i in l1:
    if i%2==0:
        l2.append(i)
print("Even numbers:",l2)
```

```
Even numbers: [2, 4, 6, 8, 10]
```

unique elements

In [3]:

```
l1=[1,2,3,4,5,1,2,3,4,5]
l2=[]
for i in l1:
    if i not in l2:
        l2.append(i)
print(l2)
print(list(set(l1)))
```

```
[1, 2, 3, 4, 5]
```

Reversing each string from a list of strings

In [5]:

```
l1=["hi","hello","racecar"]
l2=[]
for i in l1:
    l2.append(i[::-1])
print(l2)
```

```
['ih', 'olleh', 'racecar']
```

Dictionary operations and methods

In [2]:

```
# Creating a sample dictionary
sample_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Checking if a key exists in the dictionary
key_to_check = 'b'
if key_to_check in sample_dict:
    print(f'The key "{key_to_check}" exists in the dictionary.')

# Traversing the dictionary using a loop
print("Traversing the dictionary:")
for key, value in sample_dict.items():
    print(f'Key: {key}, Value: {value}')

# Dictionary methods
keys_list = list(sample_dict.keys())
values_list = list(sample_dict.values())
items_list = list(sample_dict.items())

print("\nUsing dictionary methods:")
print(f'Keys: {keys_list}')
print(f'Values: {values_list}')
print(f'Items: {items_list}')
```

```
The key "b" exists in the dictionary.
Traversing the dictionary:
Key: a, Value: 1
Key: b, Value: 2
Key: c, Value: 3
Key: d, Value: 4

Using dictionary methods:
Keys: ['a', 'b', 'c', 'd']
Values: [1, 2, 3, 4]
Items: [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

In [7]:
```
def add_entry(dictionary,key,value):
    dictionary[key]=value
    print(f"Added:{key}: {value}")
my_dictionary={"apple":"a fruit","python":"programming language","earth":"o
ur planet"}
add_entry(my_dictionary,"moon","earth's natural satellite")
print(my_dictionary)
#print(a)
#print(a,my_dictionary)
```

```
Added:moon: earth's natural satellite
{'apple': 'a fruit', 'python': 'programming language', 'earth': 'our plane
t', 'moon': "earth's natural satellite"}
```

occurrence of string

In [5]:
```
def count_occurrences(main_string, substring):
    count = 0
    start_index = 0

    while start_index < len(main_string):
        index = main_string.find(substring, start_index)

        if index == -1:
            break

        count += 1
        start_index = index + 1

    return count

# Example usage:
main_string = "abababababab ab ab"
substring = "ab"
result = count_occurrences(main_string, substring)

print(f"The substring '{substring}' occurs {result} times in the main strin
g.")
print(main_string.count(substring))
```

```
The substring 'ab' occurs 7 times in the main string.
7
```

In [2]:
```
{"a": 1, "b": 2, "c": {"d": 4, "e": 5, "f": {...}}, "abc": {...}, "abcd": 1
00}
```

Out[2]:
```
{'a': 1,
 'b': 2,
 'c': {'d': 4, 'e': 5, 'f': {Ellipsis}},
 'abc': {Ellipsis},
 'abcd': 100}
```

In [10]:

```
j1 = int(input("Capacity of jug 1: "))
j2 = int (input("Capacity of jug 2: "))
g = int (input("Amount of water to be measured: "))
def apply_rule(ch, x, y):
    # Rule 1 : Fill jug 1
    if ch == 1:
        # empty sapce in jug 1 should be less then its capacity
        if x<j1:
            return j1,y
        else:
            print("Rule cannot be applied")
            return x,y
    # Rule 2:Fill jug 2
    elif ch == 2:
    # empty sapce in jug 2 should be less then its capacity
        if y<j2:
            return x ,j2
        else:
            print("Rule cannot be applier")
            return x,y
    # Rule 3:Transfer all water from jug 1 to jug 2
    if ch == 3:
        # jug 1 should not be empty and jug 1 + jug 2 should be more then i
ts capacity of jug 2
        if x > 0 and x+y <= j2:
            return 0,x+y
        else:
            print("Rule cannot be applier")
            return x,y

    # Rule 4:Transfer all water from jug 2 to jug 1
    if ch == 4:
        # jug 2 should not be empty and jug 1 + jug 2 should not be more th
en capacity of jug 1
        if y > 0 and x+y <= j1:
            return x+y,0
        else:
            print("Rule cannot be applier")
            return x,y
    # Rule 5:Transfer some water from jug 1 to jug 2 until jug 2 is full
    if ch == 5:
        # jug 1 should not be empty and jug 1 + jug 2 should be more then o
r equal to capacity of jug 2
        if x > 0 and x+y >= j2:
            return x-(j2-y), j2
        else:
            print("Rule cannot be applier")
            return x,y
    # Rule 6:Transfer some water from jug 2 to jug 1 until jug 1 is full
    if ch == 6:
        # jug 2 should not be empty and jug 1 + jug 2 should be more then o
r equal to capacity of jug 1
        if y > 0 and x+y >= j1:
            return j1, x-(j1-x)
        else:
            print("Rule cannot be applier")
            return x,y
    # Rule 7:empty jug 1
    if ch == 7:
        # check if jug 1 is not already empty
        if x > 0:
            return 0,y
        else:
            print("Rule cannot be applier")
            return x,y
    # Rule 8:empty jug 2
    if ch == 8:
        # check if jug 2 is not already empty
        if y > 0:
            return x,0
        else:
            print("Rule cannot be applier")
            return x,y
    # invalid choice
    else:
        print("INVALID CHOICE")
# intialize capacity of both jugs as 0
x = y = 0
while(True):
    if (x==g) or (y==g):
        print('GOAL ACHIEVED!')
        break
    else:
        print("===============================RULES======================
=======")
        print("Rule 1: Fill jug 1")
        print("Rule 2: Fill jug 2")
        print("Rule 3: Transfer all water from jug 1 to jug 2")
        print("Rule 4: Transfer all water from jug 2 to jug 1")
        print("Rule 5: Transfer some water from jug 1 to jug 2 until jug 2
is full")
        print("Rule 6: Transfer some water from jug 2 to jug 1 until jug 1
is full")
        print("Rule 7: Empty jug 1")
        print("Rule 8: Empty jug 2")
        ch = int(input("Enter rule to apply: "))
        x, y = apply_rule(ch, x, y)
        print("===============================STATUS=====================
=====")
        print("CURRENT STATE:", end = " ")
        print(x, y)
```

```
================================RULES==============================
Rule 1: Fill jug 1
Rule 2: Fill jug 2
Rule 3: Transfer all water from jug 1 to jug 2
Rule 4: Transfer all water from jug 2 to jug 1
Rule 5: Transfer some water from jug 1 to jug 2 until jug 2 is full
Rule 6: Transfer some water from jug 2 to jug 1 until jug 1 is full
Rule 7: Empty jug 1
Rule 8: Empty jug 2

================================STATUS===========================
CURRENT STATE: 4 0
================================RULES==============================
Rule 1: Fill jug 1
Rule 2: Fill jug 2
Rule 3: Transfer all water from jug 1 to jug 2
Rule 4: Transfer all water from jug 2 to jug 1
Rule 5: Transfer some water from jug 1 to jug 2 until jug 2 is full
Rule 6: Transfer some water from jug 2 to jug 1 until jug 1 is full
Rule 7: Empty jug 1
Rule 8: Empty jug 2

================================STATUS===========================
CURRENT STATE: 4 3
================================RULES==============================
Rule 1: Fill jug 1
Rule 2: Fill jug 2
Rule 3: Transfer all water from jug 1 to jug 2
Rule 4: Transfer all water from jug 2 to jug 1
Rule 5: Transfer some water from jug 1 to jug 2 until jug 2 is full
Rule 6: Transfer some water from jug 2 to jug 1 until jug 1 is full
Rule 7: Empty jug 1
Rule 8: Empty jug 2

Rule cannot be applier
================================STATUS===========================
CURRENT STATE: 4 3
================================RULES==============================
Rule 1: Fill jug 1
Rule 2: Fill jug 2
Rule 3: Transfer all water from jug 1 to jug 2
Rule 4: Transfer all water from jug 2 to jug 1
Rule 5: Transfer some water from jug 1 to jug 2 until jug 2 is full
Rule 6: Transfer some water from jug 2 to jug 1 until jug 1 is full
Rule 7: Empty jug 1
Rule 8: Empty jug 2

================================STATUS===========================
CURRENT STATE: 0 3
================================RULES==============================
Rule 1: Fill jug 1
Rule 2: Fill jug 2
Rule 3: Transfer all water from jug 1 to jug 2
Rule 4: Transfer all water from jug 2 to jug 1
Rule 5: Transfer some water from jug 1 to jug 2 until jug 2 is full
Rule 6: Transfer some water from jug 2 to jug 1 until jug 1 is full
Rule 7: Empty jug 1
Rule 8: Empty jug 2
```

```
================================STATUS===========================
CURRENT STATE: 3 0
================================RULES==============================
Rule 1: Fill jug 1
Rule 2: Fill jug 2
Rule 3: Transfer all water from jug 1 to jug 2
Rule 4: Transfer all water from jug 2 to jug 1
Rule 5: Transfer some water from jug 1 to jug 2 until jug 2 is full
Rule 6: Transfer some water from jug 2 to jug 1 until jug 1 is full
Rule 7: Empty jug 1
Rule 8: Empty jug 2

================================STATUS===========================
CURRENT STATE: 3 3
================================RULES==============================
Rule 1: Fill jug 1
Rule 2: Fill jug 2
Rule 3: Transfer all water from jug 1 to jug 2
Rule 4: Transfer all water from jug 2 to jug 1
Rule 5: Transfer some water from jug 1 to jug 2 until jug 2 is full
Rule 6: Transfer some water from jug 2 to jug 1 until jug 1 is full
Rule 7: Empty jug 1
Rule 8: Empty jug 2

================================STATUS===========================
CURRENT STATE: 4 2
GOAL ACHIEVED!
```

In [1]:

```
#DFS

tree ={
1: [2,9,10],
2: [3,4],
3: [],
4: [5,6,7],
5: [8],
6: [],
7: [],
8: [],
9: [],
10: []
}
def depth_first_search(tree,start):
    stack=[start]
    visited=[]

    while stack:
        print("before",stack)
        node=stack.pop()
        visited.append(node)
        for child in reversed(tree[node]):
            if child not in visited and child not in stack:
                stack.append(child)
                print("after",stack)
    return visited
result=depth_first_search(tree,1)
print(result)
```

```
before [1]
after [10]
after [10, 9]
after [10, 9, 2]
before [10, 9, 2]
after [10, 9, 4]
after [10, 9, 4, 3]
before [10, 9, 4, 3]
before [10, 9, 4]
after [10, 9, 7]
after [10, 9, 7, 6]
after [10, 9, 7, 6, 5]
before [10, 9, 7, 6, 5]
after [10, 9, 7, 6, 8]
before [10, 9, 7, 6, 8]
before [10, 9, 7, 6]
before [10, 9, 7]
before [10, 9]
before [10]
[1, 2, 3, 4, 5, 8, 6, 7, 9, 10]
```

In [21]:

```
#BFS

tree ={
1: [2,9,10],
2: [3,4],
3: [],
4: [5,6,7],
5: [8],
6: [],
7: [],
8: [],
9: [],
10: []
}
def breadth_first_search(tree,start):
    q=[start]
    visited=[]

    while q:
        print("before",q)
        node=q.pop(0)
        visited.append(node)
        for child in (tree[node]):
            if child not in visited and child not in q:
                q.append(child)
        print("after",q)
    return visited
result=breadth_first_search(tree,1)
print(result)
```

```
before [1]
after [2, 9, 10]
before [2, 9, 10]
after [9, 10, 3, 4]
before [9, 10, 3, 4]
after [10, 3, 4]
before [10, 3, 4]
after [3, 4]
before [3, 4]
after [4]
before [4]
after [5, 6, 7]
before [5, 6, 7]
after [6, 7, 8]
before [6, 7, 8]
after [7, 8]
before [7, 8]
after [8]
before [8]
after []
[1, 2, 9, 10, 3, 4, 5, 6, 7, 8]
```

In [ ]:

In [20]:

```
def water_jug_dfs(capacity_x, capacity_y, target):
    stack = [(0, 0, [])]  # (x, y, path)
    visited_states = set()

    while stack:
        x, y, path = stack.pop()
        if (x, y) in visited_states:
            continue
        visited_states.add((x, y))
        if x == target or y == target:
            return path + [(x, y)]
        # Define possible jug operations
        operations = [
            ("fill_x", capacity_x, y),
            ("fill_y", x, capacity_y),
            ("empty_x", 0, y),
            ("empt y_y", x, 0),
            ("pour_x_to_y", max(0, x - (capacity_y - y)), min(capacity_y, y
+ x)),
            ("pour_y_to_x", min(capacity_x, x + y), max(0, y - (capacity_x
- x))),
        ]
        # print(operations)
        for operation, new_x, new_y in operations:
            if 0 <= new_x <= capacity_x and 0 <= new_y <= capacity_y:
                stack.append((new_x, new_y, path + [(x, y, operation)]))

    return None
# Example usage:
capacity_x = 4
capacity_y = 3
target = 2
solution_path = water_jug_dfs(capacity_x, capacity_y, target)
if solution_path:
    print("Solution found:")
    for state in solution_path:
        print(f"({state[0]}, {state[1]})")
else:
    print("No solution found.")
```

```
Solution found:
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

In [ ]:

In [19]:

```
#TSP
from itertools import permutations

def calculate_total_distance(tour, distances):
    total_distance = 0
    for i in range(len(tour) - 1):
        total_distance += distances[tour[i]][tour[i + 1]]
    total_distance += distances[tour[-1]][tour[0]]  # Return to the startin
g city
    # print(total_distance)
    return total_distance

def traveling_salesman_bruteforce(distances):
    cities = range(len(distances))
    min_distance = float('inf')
    optimal_tour = None

    for tour in permutations(cities):
        # print(tour)
        distance = calculate_total_distance(tour, distances)
        if distance < min_distance:
            min_distance = distance
            optimal_tour = tour
        # print(tour, distance)

    return optimal_tour, min_distance

# Example usage:
# Replace the distances matrix with your own data
distances_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

optimal_tour, min_distance = traveling_salesman_bruteforce(distances_matri
x)

print("Optimal Tour:", optimal_tour)
print("Minimum Distance:", min_distance)
```

```
Optimal Tour: (0, 1, 3, 2)
Minimum Distance: 80
```

In [ ]:

In [1]:

```python
# Tic-Tac-Toe game in Python
board = [" " for x in range(9)]
def print_board():
    row1 = "| {} | {} | {} |".format(board[0], board[1], board[2])
    row2 = "| {} | {} | {} |".format(board[3], board[4], board[5])
    row3 = "| {} | {} | {} |".format(board[6], board[7], board[8])
    print()
    print(row1)
    print(row2)
    print(row3)
    print()
def player_move(icon):
    if icon == "X":
        number = 1
    elif icon == "O":
        number = 2
    print("Your turn player {}".format(number))
    choice = int(input("Enter your move (1-9): ").strip())
    if board[choice - 1] == " ":
        board[choice - 1] = icon
    else:
        print()
        print("That space is taken!")
def is_victory(icon):
    if (board[0] == icon and board[1] == icon and board[2] == icon) or \
        (board[3] == icon and board[4] == icon and board[5] == icon) or \
        (board[6] == icon and board[7] == icon and board[8] == icon) or \
        (board[0] == icon and board[3] == icon and board[6] == icon) or \
        (board[1] == icon and board[4] == icon and board[7] == icon) or \
        (board[2] == icon and board[5] == icon and board[8] == icon) or \
        (board[0] == icon and board[4] == icon and board[8] == icon) or \
        (board[2] == icon and board[4] == icon and board[6] == icon):
        return True
    else:
        return False
def is_draw():
    if " " not in board:
        return True
    else:
        return False
while True:
    print_board()
    player_move("X")
    print_board()
    if is_victory("X"):
        print("X wins! Congratulations!")
        break
    elif is_draw():
        print("It's a draw!")
        break
    player_move("O")
    if is_victory("O"):
        print_board()
        print("O wins! Congratulations!")
        break
    elif is_draw():
        print("It's a draw!")
        break
```

```
|   |   |   |
|   |   |   |
|   |   |   |

Your turn player 1

| X |   |   |
|   |   |   |
|   |   |   |

Your turn player 2

| X | O |   |
|   |   |   |
|   |   |   |

Your turn player 1

| X | O | X |
|   |   |   |
|   |   |   |

Your turn player 2

| X | O | X |
| O |   |   |
|   |   |   |

Your turn player 1

| X | O | X |
| O | X |   |
|   |   |   |

Your turn player 2

| X | O | X |
| O | X | O |
|   |   |   |

Your turn player 1

| X | O | X |
| O | X | O |
| X |   |   |

X wins! Congratulations!
```

```
In [ ]:
```

In [18]:

```
import heapq

class priorityQueue:
    def __init__(self):
        self.cities = []
    def push(self, city, cost):
        heapq.heappush(self.cities, (cost, city))
    def pop(self):
        return heapq.heappop(self.cities)[1]
    def isEmpty(self):
        if (self.cities == []):
            return True
        else:
            return False
    def check(self):
        print(self.cities)

class ctNode:
    def __init__(self, city, distance):
        self.city = str(city)
        self.distance = str(distance)

romania = {}

def makedict():
    file = open("C:/Users/Micrp/Desktop/A_Star_Algorithm_Data/romania.txt",
'r')
    for string in file:
        line = string.split(',')
        ct1 = line[0]
        ct2 = line[1]
        dist = int(line[2])
        romania.setdefault(ct1, []).append(ctNode(ct2, dist))
        romania.setdefault(ct2, []).append(ctNode(ct1, dist))

def makehuristikdict():
    h = {}
    with open("C:/Users/Micrp/Desktop/A_Star_Algorithm_Data/romania_sld.tx
t", 'r') as file:
        for line in file:
            line = line.strip().split(",")
            node = line[0].strip()
            sld = int(line[1].strip())
            h[node] = sld
    return h

def heuristic(node, values):
    return values[node]

def astar(start, end):
    path = {}
    distance = {}
    q = priorityQueue()
    h = makehuristikdict()
    q.push(start, 0)
    distance[start] = 0
    path[start] = None
    expandedList = []
    while (q.isEmpty() == False):
        current = q.pop()
        expandedList.append(current)
        if (current == end):
            break
        for new in romania[current]:
            g_cost = distance[current] + int(new.distance)
            #print(new.city, new.distance, "now : " + str(distance[curren
t]), g_cost)
            if (new.city not in distance or g_cost < distance[new.city]):
                distance[new.city] = g_cost
                f_cost = g_cost + heuristic(new.city, h)
                #print(f_cost)
                q.push(new.city, f_cost)
                path[new.city] = current
    printoutput(start, end, path, distance, expandedList)

def printoutput(start, end, path, distance, expandedlist):
    finalpath = []
    i = end
    while (path.get(i) != None):
        finalpath.append(i)
        i = path[i]
    finalpath.append(start)
    finalpath.reverse()
    print("A-star Agorithm for Romania Map")
    print("\tArad => Bucharest")
    print("=======================================================")
    print("List of Cities that are Expanded : " + str(expandedlist))
    print("Total Number of Cities that are Expanded : " + str(len(expandedl
ist)))
    print("=======================================================")
    print("Cities in Final path : " + str(finalpath))
    print("Total Number of cities in final path are : " + str(len(finalpat
h)))
    print("Total Cost : " + str(distance[end]))

def main():
    src = "Arad"
    dst = "Bucharest"
    makedict()
    astar(src, dst)

if __name__ == "__main__" :
    main()
```

```
A-star Agorithm for Romania Map
        Arad => Bucharest
=======================================================
List of Cities that are Expanded : ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Fag
aras', 'Pitesti', 'Bucharest']
Total Number of Cities that are Expanded : 6
=======================================================
Cities in Final path : ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Buch
arest']
Total Number of cities in final path are : 5
Total Cost : 418
```

In [17]:

```python
import heapq

# Example road network graph
road_graph = {
'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},
'Zerind': {'Arad': 75, 'Oradea': 71},
'Timisoara': {'Arad': 118, 'Lugoj': 111},
'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
'Oradea': {'Zerind': 71, 'Sibiu': 151},
'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},
'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
'Drobeta': {'Mehadia': 75, 'Craiova': 120},
'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
'Bucharest': {'Fagaras': 211, 'Pitesti': 101}
}

heuristic_cost = {
    "Arad": {"Bucharest": 366},
    "Bucharest": {"Bucharest": 0},
    "Craiova": {"Bucharest": 160},
    "Dobreta": {"Bucharest": 242},
    "Eforie": {"Bucharest": 161},
    "Fagaras": {"Bucharest": 176},
    "Giurgiu": {"Bucharest": 77},
    "Hirsowa": {"Bucharest": 151},
    "Lasi": {"Bucharest": 226},
    "Lugoj": {"Bucharest": 244},
    "Mehadia": {"Bucharest": 241},
    "Neamt": {"Bucharest": 234},
    "Oradea": {"Bucharest": 380},
    "Pitesti": {"Bucharest": 100},
    "Rimnicu Vilcea": {"Bucharest": 193},
    "Sibiu": {"Bucharest": 253},
    "Timisoara": {"Bucharest": 329},
    "Urziceni": {"Bucharest": 80},
    "Vaslui": {"Bucharest": 199},
    "Zerind": {"Bucharest": 374}
}

def heuristic_cost_estimate(node, goal):
    return heuristic_cost[node][goal]

def a_star(graph, start, goal):
    open_set = [(0, start)] # Priority queue with initial node
    came_from = {}
    g_score = {city: float('inf') for city in graph}
    g_score[start] = 0

    while open_set:
        current_cost, current_city = heapq.heappop(open_set)

        if current_city == goal:
            path = reconstruct_path(came_from, goal)
            # print(graph)
            return path

        for neighbor, cost in graph[current_city].items():
            tentative_g_score = g_score[current_city] + cost
            if tentative_g_score < g_score[neighbor]:
                g_score[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic_cost_estimate(neigh
bor, goal)
                heapq.heappush(open_set, (f_score, neighbor))
                came_from[neighbor] = current_city
    return None # No path found

def reconstruct_path(came_from, current_city):
    path = [current_city]
    while current_city in came_from:
        current_city = came_from[current_city]
        path.insert(0, current_city)
    return path

def calculate_distance(graph, path):
    total_distance = 0
    for i in range(len(path)-1):
        current_city = path[i]
        next_city = path[i+1]
        total_distance += graph[current_city][next_city]
    return total_distance


start_city = 'Arad'
goal_city = 'Bucharest'


path = a_star(road_graph, start_city, goal_city)
distance = calculate_distance(road_graph, path)

print("Shortest Path from {} to {}: {}".format(start_city, goal_city, pat
h))
print("Total distance: {}".format(distance))
```

```
Shortest Path from Arad to Bucharest: ['Arad', 'Sibiu', 'Rimnicu Vilcea',
'Pitesti', 'Bucharest']
Total distance: 418
```

In [ ]:

In [15]:

```
#Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())
# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]
def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queens(N)
for i in board:
    print (i)
```

```
Enter the number of queens

[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

In [14]:

```
global facts
global rules
rules = True
facts =[["plant","mango"],["eating","mango"], ["seed","sprouts"]]
def assert_fact(fact):
    global facts
    global rules
    if not fact in facts:
        facts+=[fact]
        rules=True
while rules:
    rules=False
    for A1 in facts:
        if A1[0]=="seed":
            assert_fact(["plant",A1[1]])
        if A1[0]=="plant":
            assert_fact(["fruit",A1[1]])
        if A1[0]=="plant" and ["eating",A1[1]] in facts:
            assert_fact(["human",A1[1]])
print(facts)
```

```
[['plant', 'mango'], ['eating', 'mango'], ['seed', 'sprouts'], ['fruit', 'm
ango'], ['human', 'mango'], ['plant', 'sprouts'], ['fruit', 'sprouts']]
```

In [13]:

```python
from sympy import symbols, Not, Or, simplify

def resolve(clause1, clause2):
    """
    Resolve two clauses and return the resulting resolvent.
    """
    resolvent = []

    for literal1 in clause1:
        for literal2 in clause2:
            if literal1 == Not(literal2) or literal2 == Not(literal1):
                resolvent.extend([l for l in (clause1 + clause2) if l != literal1 and l != literal2])

    return list(set(resolvent))

def resolution(clauses):
    """
    Apply resolution to a set of clauses until no new clauses can be generated.
    """
    new_clauses = list(clauses)

    while True:
        n = len(new_clauses)
        print(new_clauses)
        print("-------------------------------------")
        pairs = [(new_clauses[i], new_clauses[j]) for i in range(n) for j in range(i+1, n)]

        for (clause1, clause2) in pairs:
            print(clause1)
            print(clause2)
            resolvent = resolve(clause1, clause2)
            print(resolvent)
            print("---------------")
            if not resolvent:
                # Empty clause found, contradiction reached
                return True

            if resolvent not in new_clauses:
                new_clauses.append(resolvent)

        if n == len(new_clauses):
            # No new clauses can be generated, exit loop
            return False

# Example usage:
if __name__ == "__main__":
    # Example clauses in CNF (Conjunctive Normal Form)
    clause1 = [symbols('P'), Not(symbols('Q'))]
    clause2 = [Not(symbols('P')), symbols('Q')]
    clause3 = [Not(symbols('P')), Not(symbols('Q'))]

    # List of clauses
    clauses = [clause1, clause2, clause3]

    result = resolution(clauses)

    if result:
        print("The set of clauses is unsatisfiable (contradiction found).")
    else:
        print("The set of clauses is satisfiable.")
```

```
[[P, ~Q], [~P, Q], [~P, ~Q]]
-------------------------------------
[P, ~Q]
[~P, Q]
[~P, ~Q, Q, P]
---------------
[P, ~Q]
[~P, ~Q]
[~Q]
---------------
[~P, Q]
[~P, ~Q]
[~P]
---------------
[[P, ~Q], [~P, Q], [~P, ~Q], [~P, ~Q, Q, P], [~Q], [~P]]
-------------------------------------
[P, ~Q]
[~P, Q]
[~P, ~Q, Q, P]
---------------
[P, ~Q]
[~P, ~Q]
[~Q]
---------------
[P, ~Q]
[~P, ~Q, Q, P]
[~P, ~Q, Q, P]
---------------
[P, ~Q]
[~Q]
[]
---------------
The set of clauses is unsatisfiable (contradiction found).
```

In [ ]: