....

//bfs

```
from collections import defaultdict
from collections import OrderedDict
from typing import Any
class Queue:
 def __init__(self):
  self.data = []

 def is_empty(self) -> bool:
  if len(self.data) == 0:
   return True
  return False

 def enqueue(self,value:Any) -> None:
  self.data.insert(0,value)

 def dequeue(self) -> Any:
  return self.data.pop()

 def peek(self) -> Any:
  if not self.is_empty():
   return self.data[-1]
  return

 def is_clear(self) -> Any:
  for i in range(len(self.data)):
   self.data.pop()
```

```
class Graph:
 def __init__(self,directed):
  self.graph = defaultdict(list)
  self.directed = directed
  self.parent = defaultdict(list)

 def add_edge(self,u,v,weight):
  if self.directed:
   value = (weight,v)
   self.graph[u].append(value)
  else:
   value = (weight,v)
   self.graph[u].append(value)
   value = (weight,u)
   self.graph[v].append(value)

 def bfs(self,current_node,goal_node):
  visited = []
  ara = []
  is_queue = []
  start_node = current_node
  queue = Queue()
  queue.enqueue((0,current_node))

  while not queue.is_empty():
   item = queue.dequeue()
   current_node = item[1]

   if current_node == goal_node:
```

```
    cost = item[0]
    print(self.parent.items())
    parent1 = OrderedDict(reversed(list(self.parent.items())))
    ara.append(goal_node)
    while start_node != goal_node:
      for key,value in parent1.items():
        if goal_node in value:
          goal_node = key
          ara.append(goal_node)

    queue.is_clear()

  else:
    if current_node in visited:
      continue

    visited.append(current_node)
    for neighbour in self.graph[current_node]:
      value = (neighbour[1])
      if neighbour[1] in is_queue:
        continue
      else:
        is_queue.append(value)
        self.parent[current_node].append(value)
        queue.enqueue((neighbour[0]+item[0],neighbour[1]))
print("Optimal Cost: ",cost)
print("Optimal path: ", end=" ")
print(ara[::-1],end=" ")
```

```
g = Graph(True)
# Inserting Starting Node, Destination, Edge Cost

g.add_edge('S', 'A', 5)
g.add_edge('S', 'B', 2)
g.add_edge('S', 'C', 4)
g.add_edge('A', 'D', 9)
g.add_edge('A', 'E', 4)
g.add_edge('D', 'H', 7)
g.add_edge('E', 'G', 6)
g.add_edge('B', 'G', 6)
g.add_edge('C', 'F', 2)
g.add_edge('F', 'G', 1)
print(g.graph)

g.bfs('S','G')
//..........................................................................................

//password_generator

import random
import array

MAX_LEN = 12
length = int(input("Enter password length: "))

DIGITS = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
LOCASE_CHARACTERS = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
                     'i', 'j', 'k', 'm', 'n', 'o', 'p', 'q',
```

```
                     'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
                     'z']

UPCASE_CHARACTERS = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
                     'I', 'J', 'K', 'M', 'N', 'O', 'P', 'Q',
                     'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y',
                     'Z']

SYMBOLS = ['@', '#', '$', '%', '=', ':', '?', '.', '/', '|', '~', '>',
           '*', '(', ')', '<']

COMBINED_LIST = DIGITS + UPCASE_CHARACTERS + LOCASE_CHARACTERS + SYMBOLS

rand_digit = random.choice(DIGITS)
rand_upper = random.choice(UPCASE_CHARACTERS)
rand_lower = random.choice(LOCASE_CHARACTERS)
rand_symbol = random.choice(SYMBOLS)

temp_pass = rand_digit + rand_upper + rand_lower + rand_symbol

for x in range(MAX_LEN - 4):
    temp_pass = temp_pass + random.choice(COMBINED_LIST)

    temp_pass_list = array.array('u', temp_pass)
    random.shuffle(temp_pass_list)

password = ""
for x in temp_pass_list:
        password = password + x
```

```
# print out password
print(password)
//........................................................................................
//bd_sequence

if __name__ == '__main__':

   birthdays = {
      'Albert Einstein': '03/14/1879',
      'Benjamin Franklin': '01/17/1706',
      'Ada Lovelace': '12/10/1815',
      'Donald Trump': '06/14/1946',
      'Rowan Atkinson': '01/6/1955'}

   print('Welcome to the birthday dictionary. We know the birthdays of:')
   for name in birthdays:
      print(name)

   print('Who\'s birthday do you want to look up?')
   name = input()
   if name in birthdays:
      print('{}\'s birthday is {}.'.format(name, birthdays[name]))
   else:
      print('Sadly, we don\'t have {}\'s birthday.'.format(name))

//.......................................................................................................

//dfs
```

```
from collections import defaultdict
from collections import OrderedDict
from typing import Any
class Stack:
 def __init__(self):
  self.data = []

 def is_empty(self) -> bool:
  if len(self.data) == 0:
   return True
  return False

 def push(self,value:Any) -> None:
  self.data.append(value)

 def pop(self) -> Any:
  return self.data.pop()

 def stack_clear(self) -> None:
  self.data.clear()

 def stack_length(self) -> Any:
  return len(self.data)

 def stack_print(self) -> None:
  print(self.data)

class Graph:
 def __init__(self,directed):
```

```
  self.graph = defaultdict(dict)

  self.directed = directed

  self.parent = defaultdict(dict)


 def add_edge(self,u,v,weight):

  if self.directed:

   self.graph[u][v] = (weight)

  else:

   self.graph[u][v] = (weight)

   self.graph[v][u] = (weight)


 def dfs(self,current_node,goal_node):

  visited = []

  ara = []

  start_node = current_node

  stack = Stack()

  stack1 = Stack()

  stack.push((current_node,0))


  while not stack.is_empty():

   item = stack.pop()

   current_node = item[0]

   print("current_node ",item[0])

   if current_node not in visited:

    stack1.push(current_node)

   stack.push((current_node,item[1]))

   print("Not Visited:")

   print("stack ",end=" ")

   stack.stack_print()
```

```
print("stack1 ",end=" ")
stack1.stack_print()

if current_node == goal_node:
  cost = item[1]
  stack.stack_clear()

else:
  if current_node in visited:
    k = stack.pop()
    print("visited")
    print("stack ",k)
    print("stack ",end=" ")
    stack.stack_print()
    k = stack1.pop()
    print("stack1",k)
    print("stack1 ",end=" ")
    stack1.stack_print()
    continue
  visited.append(current_node)
  if len(self.graph[current_node]) == 0:
    k = stack.pop()
    print("length0")
    print("stack ",k)
    print("stack ",end=" ")
    stack.stack_print()
    k = stack1.pop()
    print("stack1 ",k)
    print("stack1 ",end=" ")
```

```
        stack1.stack_print()
        continue
      for neighbour in self.graph[current_node]:
        if neighbour not in visited:
          stack.push((neighbour,self.graph[current_node][neighbour]+item[1]))
    print("Total Cost : ",cost)
    for i in range(stack1.stack_length()):
      print(stack1.pop())


g = Graph(True)
g.add_edge('S', 'A', 5)
g.add_edge('S', 'B', 2)
g.add_edge('S', 'C', 4)
g.add_edge('A', 'D', 9)
g.add_edge('A', 'E', 4)
g.add_edge('D', 'H', 7)
g.add_edge('E', 'G', 6)
g.add_edge('B', 'G', 6)
g.add_edge('C', 'F', 2)
g.add_edge('F', 'G', 1)
print(g.graph)
g.dfs('S','G')
```

....

```
//UCS
from collections import defaultdict
from queue import PriorityQueue
```

```
from collections import OrderedDict

class Graph:
 def __init__(self,directed):
  self.graph = defaultdict(dict)
  self.directed = directed
  self.parent = defaultdict(dict)

 def add_edge(self,u,v,weight):
  if self.directed:
   #value = (weight,v)
   self.graph[u][v] = (weight)
  else:
   #value = (weight,v)
   self.graph[u][v] = (weight)
   #value = (weight,u)
   self.graph[v][u] = (weight)



 def ucs(self, current_node, goal_node):
  visited = []
  ara = []
  start_node = current_node
  queue = PriorityQueue()
  queue.put((0,current_node))

  while not queue.empty():
   item = queue.get()
   current_node = item[1]
```

```
   print(current_node,item[0])

   if current_node == goal_node:
     cost = item[0]
     cost1 = cost
     ara.append(goal_node)
     print(self.parent)
     while start_node != goal_node:
       for key,value in self.parent.items():
         if goal_node in self.graph[key] and cost1==self.parent[key][goal_node]:
           cost1 = cost1 - self.graph[key][goal_node]
           ara.append(key)
           goal_node = key

     queue.queue.clear()
   else:
     if current_node in visited:
       continue

     #print(current_node, end=" ")
     visited.append(current_node)

     for neighbour in self.graph[current_node]:
       self.parent[current_node][neighbour] = (self.graph[current_node][neighbour]+item[0])
       queue.put((self.graph[current_node][neighbour]+item[0],neighbour))
 print("Optimal Cost: ",cost)
 print("Optimal path: ", end=" ")
 print(ara[::-1],end=" ")
```

```
g = Graph(True)
#g.graph = defaultdict(list)
g.add_edge('S', 'A', 5)
g.add_edge('S', 'B', 2)
g.add_edge('S', 'C', 4)
g.add_edge('A', 'D', 9)
g.add_edge('A', 'E', 4)
g.add_edge('D', 'H', 7)
g.add_edge('E', 'G', 6)
g.add_edge('B', 'G', 6)
g.add_edge('C', 'F', 2)
g.add_edge('F', 'G', 1)

print(g.graph)
g.ucs('S','G')
#print(g.parent['F']['G'])
```
……

```
//ids

from collections import defaultdict
from collections import OrderedDict
from typing import Any
class Stack:
  def __init__(self):
    self.data = []

  def is_empty(self) -> bool:
```

```
    if len(self.data) == 0:
      return True
    return False

  def push(self,value:Any) -> None:
    self.data.append(value)

  def pop(self) -> Any:
    return self.data.pop()

  def stack_clear(self) -> None:
    self.data.clear()

  def stack_length(self) -> Any:
    return len(self.data)

  def stack_print(self) -> None:
    print(self.data)

class Queue:
  def __init__(self):
    self.data = []

  def is_empty(self) -> bool:
    if len(self.data) == 0:
      return True
    return False

  def enqueue(self,value:Any) -> None:
```

```
    self.data.insert(0,value)

  def dequeue(self) -> Any:
    return self.data.pop()

  def peek(self) -> Any:
    if not self.is_empty():
      return self.data[-1]
    return

  def is_clear(self) -> Any:
    for i in range(len(self.data)):
      self.data.pop()
class Graph:
  cnt = 1
  def __init__(self,directed):
    self.graph = defaultdict(dict)
    self.directed = directed
    self.parent = defaultdict(dict)

  def add_edge(self,u,v,weight):
    if self.directed:
      self.graph[u][v] = (weight)
    else:
      self.graph[u][v] = (weight)
      self.graph[v][u] = (weight)

  def dfs(self,current_node,goal_node,stack,cnt):
    stack.stack_print()
```

```
  visited = []
  ara = []
  start_node = current_node
  temp = 0

  while not stack.is_empty():
   item = stack.pop()
   current_node = item[0]
   if current_node == goal_node:
    cost = item[1]
    temp = 1
    stack.stack_clear()

   else:
    if current_node in visited:
     continue
    visited.append(current_node)
  if temp == 1:
   print("Total Cost : ",cost)
   return -1
   #for i in range(stack1.stack_length()):
   #  print(stack1.pop())
  else:
   print(cnt)
   return cnt+1

 def ids(self,current_node,goal_node):
  visited = []
  stack = Stack()
```

```
start_node = current_node
stack.push((start_node,0,1))
stack1 = Stack()
stack1.push((start_node,0))
cnt = 1
while True:
  if stack.is_empty():
    cnt = self.dfs(start_node,goal_node,stack1,cnt)
    print(cnt)
    visited = []
    stack = Stack()
    stack.push((start_node,0,1))
    stack1 = Stack()
    stack1.push((start_node,0))
    stack.stack_print()
    stack1.stack_print()
    print(visited)
    if cnt == -1:
      return
  else:
    item = stack.pop()
    current_node = item[0]
    level = item[2]
    if current_node in visited:
      continue
    visited.append(current_node)
    for neighbour in self.graph[current_node]:
      if neighbour not in visited and level<cnt:
        if item[2] != cnt-1:
```

```
            stack.push((neighbour,self.graph[current_node][neighbour]+item[1],item[2]+1))
           stack1.push((neighbour,self.graph[current_node][neighbour]+item[1]))

g = Graph(True)
g.add_edge('S', 'A', 5)
g.add_edge('S', 'B', 2)
g.add_edge('S', 'C', 4)
g.add_edge('A', 'D', 9)
g.add_edge('A', 'E', 4)
g.add_edge('D', 'H', 7)
g.add_edge('E', 'G', 6)
g.add_edge('B', 'G', 6)
g.add_edge('C', 'F', 2)
g.add_edge('F', 'G', 1)
print(g.graph)
g.ids('S','G')
```

……

//depth_limited_search

```
from collections import defaultdict
from collections import OrderedDict
from typing import Any
class Stack:
  def __init__(self):
    self.data = []

  def is_empty(self) -> bool:
    if len(self.data) == 0:
```

```
      return True
    return False

  def push(self,value:Any) -> None:
    self.data.append(value)

  def pop(self) -> Any:
    return self.data.pop()

  def stack_clear(self) -> None:
    self.data.clear()

  def stack_length(self) -> Any:
    return len(self.data)

  def stack_print(self) -> None:
    print(self.data)

class Graph:
  def __init__(self,directed):
    self.graph = defaultdict(dict)
    self.directed = directed
    self.parent = defaultdict(dict)

  def add_edge(self,u,v,weight):
    if self.directed:
      self.graph[u][v] = (weight)
    else:
      self.graph[u][v] = (weight)
```

```
    self.graph[v][u] = (weight)

  def dfs(self,current_node,goal_node,k):
   visited = []
   ara = []
   start_node = current_node
   stack = Stack()
   stack1 = Stack()
   stack.push((current_node,0,1))

   while not stack.is_empty():
    item = stack.pop()
    current_node = item[0]
    if current_node not in visited:
     stack1.push(current_node)
    stack.push((current_node,item[1]))

    if current_node == goal_node:
     cost = item[1]
     stack.stack_clear()

    else:
     if current_node in visited:
      stack.pop()
      stack1.pop()
      continue

     visited.append(current_node)
     if len(self.graph[current_node]) == 0:
```

```
          stack.pop()
          stack1.pop()
          continue
        for neighbour in self.graph[current_node]:
          if neighbour not in visited and item[2]<k:
            stack.push((neighbour,self.graph[current_node][neighbour]+item[1],item[2]+1))
    print("Total Cost : ",cost)
    for i in range(stack1.stack_length()):
      print(stack1.pop())
g = Graph(True)
g.add_edge('S', 'A', 5)
g.add_edge('S', 'B', 2)
g.add_edge('S', 'C', 4)
g.add_edge('A', 'D', 9)
g.add_edge('A', 'E', 4)
g.add_edge('D', 'H', 7)
g.add_edge('E', 'G', 6)
g.add_edge('B', 'G', 6)
g.add_edge('C', 'F', 2)
g.add_edge('F', 'G', 1)
print(g.graph)
k = int(input("Enter depth Limit : "))
g.dfs('S','G',k)
```

....

//bidirectional search

```
from collections import defaultdict
from collections import OrderedDict
```

```
class Graph:
 def __init__(self):
  self.graph1 = defaultdict(dict)
  self.graph2 = defaultdict(dict)
  self.parent = defaultdict(dict)

 def add_edge(self,u,v,weight):
  self.graph1[u][v] = (weight)
  self.graph2[v][u] = (weight)

 def bidirectional(self,current_node,goal_node):
  visited1 = []
  visited2 = []
  queue = []
  stack = []

  queue.append((current_node,0))
  stack.append((goal_node,0))

  while len(queue) != 0:

   # For BFS
   item1 = queue.pop(0)
   current_node1 = item1[0]
   print("BFS : ",current_node1)

   if current_node1 in visited2:
    print("Goal Node is Found")
    queue.clear()
```

```
  stack.clear()
  continue
else:
  if current_node1 in visited1:
    continue

  visited1.append(current_node1)

  for neighbour1 in self.graph1[current_node1]:
    queue.append((neighbour1,self.graph1[current_node1][neighbour1]))

# For DFS

item2 = stack.pop(-1)
current_node2 = item2[0]
print("DFS : ", current_node2)

if current_node2 in visited1:
  print("Goal node is found")
  queue.clear()
  stack.clear()
  continue

else:
  if current_node2 in visited2:
    continue

  visited2.append(current_node2)
```

```
        for neighbour2 in self.graph2[current_node2]:
          stack.append((neighbour2,self.graph2[current_node2][neighbour2]))
g = Graph()
g.graph1
g.graph2
g.add_edge('S', 'A', 5)
g.add_edge('S', 'B', 2)
g.add_edge('S', 'C', 4)
g.add_edge('A', 'D', 9)
g.add_edge('A', 'E', 4)
g.add_edge('D', 'H', 7)
g.add_edge('E', 'G', 6)
g.add_edge('B', 'G', 6)
g.add_edge('C', 'F', 2)
g.add_edge('F', 'G', 1)

g.bidirectional('S','G')
```

....

```
//greedy_best first search

from collections import defaultdict
from queue import PriorityQueue
from collections import OrderedDict
class Graph:
  def __init__(self,directed):
    self.graph = defaultdict(dict)
    self.directed = directed
    self.parent = defaultdict(dict)
```

```
  self.heuristic = defaultdict(dict)

def add_edge(self,u,v,weight):
  if self.directed:
    #value = (weight,v)
    self.graph[u][v] = (weight)
  else:
    #value = (weight,v)
    self.graph[u][v] = (weight)
    #value = (weight,u)
    self.graph[v][u] = (weight)

def add_heuristic(self,node,value):
  self.heuristic[node] = (value)



def ucs(self, current_node, goal_node):
  visited = []
  ara = []
  start_node = current_node
  h_value = self.heuristic[current_node]
  queue = PriorityQueue()
  queue.put((h_value,current_node,0))

  while not queue.empty():
    item = queue.get()
    current_node = item[1]
    print(current_node,item[2])
```

```
        if current_node == goal_node:
          cost = item[2]
          cost1 = cost
          ara.append(goal_node)
          print(self.parent)
          while start_node != goal_node:
            for key,value in self.parent.items():
              if goal_node in self.graph[key] and cost1==self.parent[key][goal_node]:
                cost1 = cost1 - self.graph[key][goal_node]
                ara.append(key)
                goal_node = key

          queue.queue.clear()
        else:
          if current_node in visited:
            continue

          #print(current_node, end=" ")
          visited.append(current_node)

          for neighbour in self.graph[current_node]:
            self.parent[current_node][neighbour] = (self.graph[current_node][neighbour]+item[2])
            queue.put((self.heuristic[neighbour],neighbour,self.graph[current_node][neighbour]+item[2]))
    print("Optimal Cost: ",cost)
    print("Optimal path: ", end=" ")
    print(ara[::-1],end=" ")
g = Graph(True)
    #g.graph = defaultdict(list)
#g.add_edge('S', 'A', 5)
```

```
#g.add_edge('S', 'B', 2)
#g.add_edge('S', 'C', 4)
#g.add_edge('A', 'D', 9)
#g.add_edge('A', 'E', 4)
#g.add_edge('D', 'H', 7)
#g.add_edge('E', 'G', 6)
#g.add_edge('B', 'G', 6)
#g.add_edge('C', 'F', 2)
#g.add_edge('F', 'G', 1)
infinity = 100000009

g.add_heuristic('S',8)
g.add_heuristic('A',8)
g.add_heuristic('B',4)
g.add_heuristic('C',3)
g.add_heuristic('D',infinity)
g.add_heuristic('E',infinity)
g.add_heuristic('G',0)


g.add_edge('S','A',1)
g.add_edge('S','B',5)
g.add_edge('S','C',8)
g.add_edge('A','D',3)
g.add_edge('A','E',7)
g.add_edge('A','G',9)
g.add_edge('B','G',4)
g.add_edge('C','G',5)
print(g.graph)
```

```
print(g.heuristic)
g.ucs('S','G')
```

………

```
//beam search
from collections import defaultdict
from queue import PriorityQueue
from collections import OrderedDict

class Graph:
  def __init__(self,directed):
    self.graph = defaultdict(dict)
    self.directed = directed
    self.parent = defaultdict(dict)
    self.heuristic = defaultdict(dict)

  def add_edge(self,u,v,weight):
    if self.directed:
      #value = (weight,v)
      self.graph[u][v] = (weight)
    else:
      #value = (weight,v)
      self.graph[u][v] = (weight)
      #value = (weight,u)
      self.graph[v][u] = (weight)

  def add_heuristic(self,node,value):
```

```
 self.heuristic[node] = (value)



def ucs(self, current_node, goal_node,k):
 visited = []
 ara = []
 start_node = current_node
 h_value = self.heuristic[current_node]
 queue = []
 queue.append((h_value,current_node,0))
 queue.sort()

 while len(queue) != 0:
  print(queue)
  item = queue.pop(0)
  current_node = item[1]
  print(current_node,item[2])

  if current_node == goal_node:
   cost = item[2]
   cost1 = cost
   ara.append(goal_node)
   print(self.parent)
   while start_node != goal_node:
    for key,value in self.parent.items():
     if goal_node in self.graph[key] and cost1==self.parent[key][goal_node]:
      cost1 = cost1 - self.graph[key][goal_node]
      ara.append(key)
      goal_node = key
```

```
        queue.clear()
      else:
        if current_node in visited:
          continue

        #print(current_node, end=" ")
        visited.append(current_node)

        for neighbour in self.graph[current_node]:
          self.parent[current_node][neighbour] = (self.graph[current_node][neighbour]+item[2])
          queue.append((self.heuristic[neighbour],neighbour,self.graph[current_node][neighbour]+item[2]))
          queue.sort()
          if(len(queue)>k):
            queue.pop(-1)

    print("Optimal Cost: ",cost)
    print("Optimal path: ", end=" ")
    print(ara[::-1],end=" ")

g = Graph(True)
#g.graph = defaultdict(list)
#g.add_edge('S', 'A', 5)
#g.add_edge('S', 'B', 2)
#g.add_edge('S', 'C', 4)
#g.add_edge('A', 'D', 9)
#g.add_edge('A', 'E', 4)
#g.add_edge('D', 'H', 7)
#g.add_edge('E', 'G', 6)
```

```
#g.add_edge('B', 'G', 6)
#g.add_edge('C', 'F', 2)
#g.add_edge('F', 'G', 1)
infinity = 100000009

g.add_heuristic('S',8)
g.add_heuristic('A',8)
g.add_heuristic('B',4)
g.add_heuristic('C',3)
g.add_heuristic('D',infinity)
g.add_heuristic('E',infinity)
g.add_heuristic('G',0)


g.add_edge('S','A',1)
g.add_edge('S','B',5)
g.add_edge('S','C',8)
g.add_edge('A','D',3)
g.add_edge('A','E',7)
g.add_edge('A','G',9)
g.add_edge('B','G',4)
g.add_edge('C','G',5)

print(g.graph)
print(g.heuristic)
k = int(input("Enter the value of k : "))

g.ucs('S','G',k)
visited = []
```

```
visited.append((1,2))
visited.append((2,3))
visited.append((4,5))
print(visited[1][1])
visited.pop(0)
print(visited)
print(len(visited))
visited.sort(reverse=True)
print(visited)
visited.clear()
print(visited)
```

....

```
// A search algo

from collections import defaultdict
from collections import OrderedDict
from queue import PriorityQueue
class Graph:
  def __init__(self,directed):
    self.graph = defaultdict(dict)
    self.directed = directed
    self.parent = defaultdict(dict)
    self.heuristic = defaultdict(dict)

  def add_edge(self,u,v,weight):
    if self.directed:
      #value = (weight,v)
      self.graph[u][v] = (weight)
```

```
  else:
    #value = (weight,v)
    self.graph[u][v] = (weight)
    #value = (weight,u)
    self.graph[v][u] = (weight)

def add_heuristic(self,node,value):
  self.heuristic[node] = (value)



def ucs(self, current_node, goal_node):
  visited = []
  ara = []
  start_node = current_node
  h_value = self.heuristic[current_node]
  e_function = 0 + h_value
  queue = PriorityQueue()
  queue.put((e_function,current_node,0))

  while not queue.empty():
    item = queue.get()
    current_node = item[1]
    print(current_node,item[2])

    if current_node == goal_node:
      cost = item[2]
      cost1 = cost
      ara.append(goal_node)
      print(self.parent)
```

```
      while start_node != goal_node:
        for key,value in self.parent.items():
          if goal_node in self.graph[key] and cost1==self.parent[key][goal_node]:
            cost1 = cost1 - self.graph[key][goal_node]
            ara.append(key)
            goal_node = key

      queue.queue.clear()
    else:
      if current_node in visited:
        continue

      #print(current_node, end=" ")
      visited.append(current_node)

      for neighbour in self.graph[current_node]:
        val = self.graph[current_node][neighbour]+item[2]
        self.parent[current_node][neighbour] = (val)
        queue.put((self.heuristic[neighbour]+val,neighbour,val))
  print("Optimal Cost: ",cost)
  print("Optimal path: ", end=" ")
  print(ara[::-1],end=" ")

g = Graph(True)
#g.graph = defaultdict(list)
#g.add_edge('S', 'A', 5)
#g.add_edge('S', 'B', 2)
#g.add_edge('S', 'C', 4)
#g.add_edge('A', 'D', 9)
```

```
#g.add_edge('A', 'E', 4)
#g.add_edge('D', 'H', 7)
#g.add_edge('E', 'G', 6)
#g.add_edge('B', 'G', 6)
#g.add_edge('C', 'F', 2)
#g.add_edge('F', 'G', 1)
infinity = 100000009

g.add_heuristic('S',8)
g.add_heuristic('A',8)
g.add_heuristic('B',4)
g.add_heuristic('C',3)
g.add_heuristic('D',infinity)
g.add_heuristic('E',infinity)
g.add_heuristic('G',0)



g.add_edge('S','A',1)
g.add_edge('S','B',5)
g.add_edge('S','C',8)
g.add_edge('A','D',3)
g.add_edge('A','E',7)
g.add_edge('A','G',9)
g.add_edge('B','G',4)
g.add_edge('C','G',5)

print(g.graph)
print(g.heuristic)
g.ucs('S','G')
```

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

print(q.pop(2))

print(q)
```

………

```
//hill_climbing

SuccList ={ 'A':[['B',3],['C',2]], 'B':[['D',2],['E',3]], 'C':[['F',2],['G',4]], 'D':[['H',1],['I',99]],'F': [['J',1]]
,'G':[['K',99],['L',3]]}

Start='A'

Closed = list()

SUCCESS=True

FAILURE=False

def MOVEGEN(N):
	New_list=list()
	if N in SuccList.keys():
		New_list=SuccList[N]

	return New_list

def SORT(L):
	L.sort(key = lambda x: x[1])
	return L

def heu(Node): #Node = ['B',2]--> [Node[0],Node[1]]
	return Node[1]
```

```
def APPEND(L1,L2):
    New_list=list(L1)+list(L2)
    return New_list

def Hill_Climbing(Start):
    global Closed
    N=Start
    CHILD = MOVEGEN(N)
    SORT(CHILD)
    N=[Start,5]
    print("\nStart=",N)
    print("Sorted Child List=",CHILD)
    newNode=CHILD[0]
    CLOSED=[N]

    while (heu(newNode) < heu(N)) and (len(CHILD) !=0):
        print("\n--------------------------")
        N= newNode
        print("N=",N)
        CLOSED = APPEND(CLOSED,[N])
        CHILD = MOVEGEN(N[0])
        SORT(CHILD)
        print("Sorted Child List=",CHILD)
        print("CLOSED=",CLOSED)
        newNode=CHILD[0]

    Closed=CLOSED
```

```
#Driver Code
Hill_Climbing(Start) #call search algorithm
```

....

```
// hill climbing TSP
import random
def randomSolution(tsp):
  cities = list(range(len(tsp)))
  #print(cities)
  solution = []

  for i in range(len(tsp)):
    randomCity = cities[random.randint(0,len(cities)-1)]
    solution.append(randomCity)
    cities.remove(randomCity)

  return solution
def routeLength(tsp,solution):
  routeLength = 0
  for i in range(len(solution)):
    routeLength += tsp[solution[i-1]][solution[i]]

  return routeLength

def getNeighbours(solution):
  neighbours = []
  for i in range(len(solution)):
    for j in range(i+1,len(solution)):
```

```
      neighbour = solution.copy()
      #print(neighbour)
      neighbour[i] = solution[j]
      neighbour[j] = solution[i]
      #print(neighbour)
      #print('HI')
      neighbours.append(neighbour)

  return neighbours

def getBestNeighbour(tsp, neighbours):
  bestRouteLength = routeLength(tsp,neighbours[0])
  bestNeighbour = neighbours[0]
  for neighbour in neighbours:
    currentRouteLength = routeLength(tsp,neighbour)
    if currentRouteLength < bestRouteLength:
      bestRouteLength = currentRouteLength
      bestNeighbour = neighbour

  return bestNeighbour, bestRouteLength

def hillClimbing(tsp):
  currentSolution = randomSolution(tsp)
  currentRouteLength = routeLength(tsp,currentSolution)

  neighbours = getNeighbours(currentSolution)
  bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp,neighbours)

  while bestNeighbourRouteLength < currentRouteLength:
```

```
    currentSolution = bestNeighbour
    currentRouteLength = bestNeighbourRouteLength
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

  return currentSolution, currentRouteLength

def main():
  tsp = [
      [0,400,500,300],
      [400,0,300,500],
      [500,300,0,400],
      [300,500,400,0]
  ]

  #solution = [3,0,1,2]
  #print(getNeighbours(solution))

  #print(randomSolution(tsp))

  print(hillClimbing(tsp))



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

....

```
//simulated_anneling
import random,sys,math

SuccList ={ 'A':[['T',11],['B',13],['C',21]],
                'T':[['D',27],['B',13]],
                'B':[['D',27],['E',3]],
                'C':[['F',25],['G',4]],
                'D':[['H',101],['I',99]],
                'F': [['J',67]],
                'G':[['K',99],['L',3]],
                'H':[['M',17]],
                'I':[['M',17]],
                'J':[['M',17]]
                }
Start='A'

Closed = list()
def MOVEGEN(N):
	New_list=list()
	if N in SuccList.keys():
		New_list=SuccList[N]

	return New_list

def heu(Node): #Node = ['B',2]--> [Node[0],Node[1]]
	return Node[1]

def APPEND(L1,L2):
	New_list=list(L1)+list(L2)
```

```
        return New_list


def exp_schedule(k=20, lam=0.005, limit=100):
        #k is Boltzman constant, lambda is an arbitrary constant, limit is max temperate limit
    """One possible schedule function for simulated annealing"""
    return lambda t: (k * math.exp(-lam * t) if t < limit else 0)


def Simulated_Annealing(schedule=exp_schedule()):
        global Closed
        node=[Start,5]
        bestNode=node
        CLOSED=[node]

        print("\nInitial Point=",node)
        for t in range(sys.maxsize):
                print("\n*******************************")
                print("For iteration i=",t)
                T=schedule(t)
                if T == 0:
                        return bestNode

                CHILD = MOVEGEN(node[0])
                if(len(CHILD) == 0):
                        return bestNode

                node=random.choice(CHILD)
                CLOSED=APPEND(CLOSED,[node])
                print("\nCurrent node=",node)
                print("Closed List=",CLOSED)
```

```
            delta_e = heu(node) - heu(bestNode)
            #The continuous uniform distribution or rectangular distribution, choice(0,1)
            #describes an experiment where there is an arbitrary outcome that lies between certain
bounds
                                          #probability of making move from currentstate
to new state
            if delta_e >0 and 1/(1+(math.exp(delta_e / T)))>random.choice([0,1]) :
                bestNode=node

    Closed=CLOSED
    return bestNode

#Driver Code
bestNode=Simulated_Annealing() #call search algorithm
print("Best Node=",bestNode)
```

....

```
//fibonaccy
# 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
```

....

///merge sort

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
```

```
for i in range(0, n1):
    L[i] = arr[l + i]

for j in range(0, n2):
    R[j] = arr[m + 1 + j]

# Merge the temp arrays back into arr[l..r]
i = 0     # Initial index of first subarray
j = 0     # Initial index of second subarray
k = l     # Initial index of merged subarray

while i < n1 and j < n2:
    if L[i] <= R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1

# Copy the remaining elements of L[], if there
# are any
while i < n1:
    arr[k] = L[i]
    i += 1
    k += 1

# Copy the remaining elements of R[], if there
# are any
```

```
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted


def mergeSort(arr, l, r):
    if l < r:

        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)


# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print("Given array is")
for i in range(n):
    print("%d" % arr[i],end=" ")
```

```
mergeSort(arr, 0, n-1)
print("\n\nSorted array is")
for i in range(n):
    print("%d" % arr[i],end=" ")
```
....

```
//factorial

num = int(input("Enter a number: "))
factorial = 1
if num < 0:
   print(" Factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1,num + 1):
       factorial = factorial*i
   print("The factorial of",num,"is",factorial)
```
....

```
//lcm

# Python Program to find the L.C.M. of two input number

def compute_lcm(x, y):

   # choose the greater number
   if x > y:
       greater = x
```

```
  else:
    greater = y

  while(True):
    if((greater % x == 0) and (greater % y == 0)):
      lcm = greater
      break
    greater += 1

  return lcm

num1 = 54
num2 = 24

print("The L.C.M. is", compute_lcm(num1, num2))
```

……………………………………………………………………………………………