# **randomized quick sort**

import random

'''

The function which implements QuickSort.

arr :- array to be sorted.

start :- starting index of the array.

stop :- ending index of the array.

'''

def quicksort(arr, start , stop):

if(start < stop):

# pivotindex is the index where

# the pivot lies in the array

pivotindex = partitionrand(arr,\

start, stop)

# At this stage the array is

# partially sorted around the pivot.

# Separately sorting the

# left half of the array and the

# right half of the array.

quicksort(arr , start , pivotindex-1)

quicksort(arr, pivotindex + 1, stop)

# This function generates random pivot,

# swaps the first element with the pivot

# and calls the partition function.

def partitionrand(arr , start, stop):

# Generating a random number between the

# starting index of the array and the

# ending index of the array.

randpivot = random.randrange(start, stop)

# Swapping the starting element of

# the array and the pivot

arr[start], arr[randpivot] = \

arr[randpivot], arr[start]

return partition(arr, start, stop)

'''

This function takes the first element as pivot,

places the pivot element at the correct position

in the sorted array. All the elements are re-arranged

according to the pivot, the elements smaller than the

pivot is places on the left and the elements

greater than the pivot is placed to the right of pivot.

'''

def partition(arr,start,stop):

pivot = start # pivot

# a variable to memorize where the

i = start + 1

# partition in the array starts from.

for j in range(start + 1, stop + 1):

# if the current element is smaller

# or equal to pivot, shift it to the

# left side of the partition.

if arr[j] <= arr[pivot]:

arr[i] , arr[j] = arr[j] , arr[i]

i = i + 1

arr[pivot] , arr[i - 1] =\

arr[i - 1] , arr[pivot]

pivot = i - 1

return (pivot)

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

array = [10, 7, 8, 9, 1, 5]

quicksort(array, 0, len(array) - 1)

print(array)

# **monte carlo primality testing**

import random

# Iterative Function to calculate

# (a^n)%p in O(logy)

def power(a, n, p):

# Initialize result

res = 1

# Update 'a' if 'a' >= p

a = a % p

while n > 0:

# If n is odd, multiply

# 'a' with result

if n % 2:

res = (res \* a) % p

n = n - 1

else:

a = (a \*\* 2) % p

# n must be even now

n = n // 2

return res % p

# If n is prime, then always returns true,

# If n is composite than returns false with

# high probability Higher value of k increases

# probability of correct result

def isPrime(n, k):

# Corner cases

if n == 1 or n == 4:

return False

elif n == 2 or n == 3:

return True

# Try k times

else:

for i in range(k):

# Pick a random number

# in [2..n-2]

# Above corner cases make

# sure that n > 4

a = random.randint(2, n - 2)

# Fermat's little theorem

if power(a, n - 1, n) != 1:

return False

return True

# Driver code

k = 3

if isPrime(11, k):

print("true")

else:

print("false")

if isPrime(15, k):

print("true")

else:

print("false")

# **A\***

tree = {'S': [['A', 1], ['B', 5], ['C', 8]],

'A': [['S', 1], ['D', 3], ['E', 7], ['G', 9]],

'B': [['S', 5], ['G', 4]],

'C': [['S', 8], ['G', 5]],

'D': [['A', 3]],

'E': [['A', 7]]}

heuristic = {'S': 8, 'A': 8, 'B': 4, 'C': 3, 'D': 5000, 'E': 5000, 'G': 0}

cost = {'S': 0} # total cost for nodes visited

def AStarSearch():

global tree, heuristic

closed = [] # closed nodes

opened = [['S', 8]] # opened nodes

'''find the visited nodes'''

while True:

fn = [i[1] for i in opened] # fn = f(n) = g(n) + h(n)

chosen\_index = fn.index(min(fn))

node = opened[chosen\_index][0] # current node

closed.append(opened[chosen\_index])

del opened[chosen\_index]

if closed[-1][0] == 'G': # break the loop if node G has been found

break

for item in tree[node]:

if item[0] in [closed\_item[0] for closed\_item in closed]:

continue

cost.update({item[0]: cost[node] + item[1]}) # add nodes to cost dictionary

fn\_node = cost[node] + heuristic[item[0]] + item[1] # calculate f(n) of current node

temp = [item[0], fn\_node]

opened.append(temp) # store f(n) of current node in array

opened

'''find optimal sequence'''

trace\_node = 'G' # correct optimal tracing node, initialize as node G

optimal\_sequence = ['G'] # optimal node sequence

for i in range(len(closed)-2, -1, -1):

check\_node = closed[i][0] # current node

if trace\_node in [children[0] for children in tree[check\_node]]:

children\_costs = [temp[1] for temp in tree[check\_node]]

children\_nodes = [temp[0] for temp in tree[check\_node]]

'''check whether h(s) + g(s) = f(s). If so, append current node to optimal sequence

change the correct optimal tracing node to current node'''S.THARUN

19MID0031

if cost[check\_node] + children\_costs[children\_nodes.index(trace\_node)] ==

cost[trace\_node]:

optimal\_sequence.append(check\_node)

trace\_node = check\_node

optimal\_sequence.reverse() # reverse the optimal sequence

return closed, optimal\_sequence

if \_name\_ == '\_main\_':

visited\_nodes, optimal\_nodes = AStarSearch()

print('visited nodes: ' + str(visited\_nodes))

print('optimal nodes sequence: ' + str(optimal\_nodes))

# **AO\***