**ĐẠI HỌC HUẾ**



# KHOA KỸ THUẬT VÀ CÔNG NGHỆ

🙠🙟🕮🙝🙢

****

**BÁO CÁO**

**ĐỒ ÁN**

**NĂM HỌC 2020-2021**

**Giáo viên hướng dẫn: HỒ QUỐC DŨNG**

**Lớp: KHDL & TTNT**

|  |
| --- |
| Số phách  *(Do hội đồng chấm thi ghi)* |

**Thừa Thiên Huế, ngày 28 tháng 6 năm 2021**

# ĐẠI HỌC HUẾ



# KHOA KỸ THUẬT VÀ CÔNG NGHỆ

🙠🙟🕮🙝🙢

****

(MẪU BÌA PHỤ)

**BÁO CÁO**

**ĐỒ ÁN**

**NĂM HỌC 2020-2021**

**Giảng viên hướng dẫn: HỒ QUỐC DŨNG**

**Lớp: KHDL & TTNT**

**Sinh viên thực hiện: NGUYỄN TẤT DUY THÀNH**

*(ký tên và ghi rõ họ tên)*

|  |
| --- |
| Số phách  *(Do hội đồng chấm thi ghi)* |

**Thừa Thiên Huế, ngày 28 tháng 6 năm 2021**

**I. Tháp Hà Nội**

**1.1 R**

thap\_ha\_noi <- function(n, toa1, toa2, toa3){

if (n==1){

print(toa1)

print('|')

print(toa3)

print('')

} else {

thap\_ha\_noi(n-1,toa1, toa3, toa2)

print(toa1)

print('|')

print(toa3)

print('')

thap\_ha\_noi(n-1,toa2, toa1, toa3)

}

}

thap\_ha\_noi(3,'A','B','C')

**1.2 Python**

def Thaphanoi(n , vt1, vt2, a):

if n==1:

print ("Chuyển đĩa 1 từ vị trí",vt1,"đến vị trí",vt2 )

return

Thaphanoi(n-1, vt1, a, vt2)

print ("Chuyển đĩa",n,"tự vị trí",vt1,"đến vị trí",vt2 )

Thaphanoi(n-1, a, vt2, vt1)

n = 4

Thaphanoi(n,'A','B','C')

**II. Ước chung lớn nhất**

**2.1 R**

ucln <- function(a, b){

if(b == 0){

return(a)

} else {

return(ucln(b, a%%b))

}

}

ucln(15,10)

**2.2 Python**

def uscln(a, b):

if (b == 0):

return a

return uscln(b, a % b)

a = int(input("Số dương thứ nhất: "))

b = int(input("Số dương thứ hai: "))

print("Ước số chung lớn nhất của", a, "và", b, "là:", uscln(a, b))

**III. Tính giai thừa**

**3.1 R**

giai\_thua <- function(n){

if(n == 0){

return(1)

} else {

return(n\*giai\_thua(n-1))

}

}

giai\_thua(3)

**3.2 Python**

def giaiThua(n):

if n == 0:

return 1

return n \* giaiThua(n - 1)

n = int(input("Nhập số cần tính giai thừa: "))

print (giaiThua(n))

**IV. Mã đi tuần**

**4.1 R**

install.packages("purrr")

install.packages("dplyr")

install.packages("tidyverse")

library(purrr)

library(dplyr)

library(tidyverse)

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Knight offsets i.e. the possible movements of a knight from the current location

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

knight\_offsets <- matrix(c(

1, 2,

2, 1,

-2, 1,

-1, 2,

2, -1,

1, -2,

-1, -2,

-2, -1

), ncol = 2, byrow = TRUE)

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#' Recurisvely calculate moves for a knight to complete a tour

#'

#' @param this\_move proposed next move. 2 element numeric vector of (row, col)

#' position at which to place the knight next

#' @param moves list of vectors. Each vector is length=2 and indicates the

#' row/column locations of the knight's tour so far

#' @param visited 8x8 logical matrix which indicates whether or not a square has been

#' visited by the knight already. When called by the user, this matrix

#' must only contain FALSE

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

move\_knight <- function(this\_move, moves, visited) {

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Mark the move as visited

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

moves <- append(moves, list(this\_move))

visited[this\_move[1] + (this\_move[2] - 1)\*8] <- TRUE

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# termination if all visited

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (all(visited)) {

return(moves)

}

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Find all possible moves from this position

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

next\_move <- cbind(knight\_offsets[,1] + this\_move[1], knight\_offsets[,2] + this\_move[2])

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# keep only moves that remain on the board

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

on\_board <- next\_move[,1] %in% 1:8 & next\_move[,2] %in% 1:8

next\_move <- next\_move[on\_board,,drop=FALSE]

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Keep only moves that target a location that has not yet been visited

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

not\_yet\_visited <- !visited[next\_move]

next\_move <- next\_move[not\_yet\_visited,, drop = FALSE]

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Recurse over every possible next move

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

for (i in seq\_len(nrow(next\_move))) {

res <- move\_knight(next\_move[i,, drop = FALSE], moves, visited)

if (!is.null(res)) {

return(res)

}

}

NULL

}

system.time({

moves <- move\_knight(c(4, 8), moves = list(), visited = matrix(FALSE, 8, 8))

})

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#' Convert results to a data.frame for ggplot

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

moves\_df <- as.data.frame(do.call(rbind, moves))

moves\_df <- set\_names(moves\_df, c('x', 'y'))

moves\_df$idx <- 1:nrow(moves\_df)

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Plot the knight's tour

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ggplot(moves\_df, aes(x, y)) +

geom\_tile(aes(fill=as.logical((x+y)%%2)), colour = 'black') +

geom\_path(alpha = 0.7, linetype = 1, size = 0.25) +

geom\_text(aes(label = idx)) +

scale\_fill\_manual(values = c('grey70', 'white')) +

theme\_void() +

theme(legend.position = 'none') +

coord\_equal() +

labs(

title = "A knight's tour in #RStats"

)

**4.2 Python**

n=7

def isSafe(x, y, board):

if(x >= 0 and y >= 0 and x < n and y < n and board[x][y] == -1):

return True

return False

def printSolution(n, board):

for i in range(n):

for j in range(n):

print(board[i][j], end=' ')

print()

def solveKT(n):

board = [[-1 for i in range(n)]for i in range(n)]

move\_x = [2, 1, -1, -2, -2, -1, 1, 2]

move\_y = [1, 2, 2, 1, -1, -2, -2, -1]

board[0][0] = 0

pos = 1

if(not solveKTUtil(n, board, 0, 0, move\_x, move\_y, pos)):

print("Solution does not exist")

else:

printSolution(n, board)

def solveKTUtil(n, board, curr\_x, curr\_y, move\_x, move\_y, pos):

if(pos == n\*\*2):

return True

for i in range(8):

new\_x = curr\_x + move\_x[i]

new\_y = curr\_y + move\_y[i]

if(isSafe(new\_x, new\_y, board)):

board[new\_x][new\_y] = pos

if(solveKTUtil(n, board, new\_x, new\_y, move\_x, move\_y, pos+1)):

return True

board[new\_x][new\_y] = -1

return False

solveKT(n)

**V. Tám Quân Hậu**

**5.1 R**

library(tidyverse)

#-----------------------------------------------------------------------------

#' Try and place a Queen given a vector of positions of the current Queens

#'

#' This function calls itself recursively for every valid placement of the

#' next queen.

#'

#' @param queens A vector of integers representing the column placement

#' of queens so far. The index within this list

#' is the row, and the value is the column.

#' To generate all solutions, pass in an empty vector (the default)

#'

#' e.g. queens = c(1, 4, 7) corresponds to queens placed at c(1, 1), c(2, 4) and

#' c(3, 7)

#'

#'

#' ---------------------------------

#' | | | | | | | | |

#' ---------------------------------

#' | | | | | | | | |

#' ---------------------------------

#' | | | | | | | | |

#' ---------------------------------

#' | | | | | | | | |

#' ---------------------------------

#' | | | | | | | Q | | 3rd row, 7th column

#' ---------------------------------

#' | | | | Q | | | | | 2nd row, 4th column

#' ---------------------------------

#' | Q | | | | | | | | 1st row, 1st column

#' ---------------------------------

#'

#'

#'

#'

#' @return a list where each element is a vector of 8 integers

#' i.e. a solution to the 8 queens problem

#-----------------------------------------------------------------------------

place\_queen <- function(queens=c()) {

#---------------------------------------------------------------------------

# If there are 8 queens placed, then this must be a solution.

#---------------------------------------------------------------------------

if (length(queens) == 8) {

return(list(queens))

}

#---------------------------------------------------------------------------

# Figure out where a queen can be placed in the next row.

# Drop all columns that have already been taken - since we

# can't place a queen below an existing queen

#---------------------------------------------------------------------------

possible\_placements <- setdiff(1:8, queens)

#---------------------------------------------------------------------------

# For each queen already on the board, find the diagonal

# positions that it can see in this row.

#---------------------------------------------------------------------------

diag\_offsets <- seq.int(length(queens), 1)

diags <- c(queens + diag\_offsets, queens - diag\_offsets)

diags <- diags[diags > 0 & diags < 9]

#---------------------------------------------------------------------------

# Drop these diagonal columns from possible placements

#---------------------------------------------------------------------------

possible\_placements <- setdiff(possible\_placements, diags)

#---------------------------------------------------------------------------

# For each possible placement, try and place a queen

#---------------------------------------------------------------------------

possible\_placements %>%

map(~place\_queen(c(queens, .x))) %>%

keep(~length(.x) > 0) %>%

flatten()

}

#-----------------------------------------------------------------------------

#' Plot a single solution

#' @param queens a vector of 8 integers giving the column positions of 8 queens

#-----------------------------------------------------------------------------

plot\_single\_8queens <- function(queens, title = NULL) {

queens\_df <- tibble(cols = queens, rows=1:8)

board\_df <- expand.grid(cols = 1:8, rows = 1:8) %>%

mutate(check = (cols + rows) %%2 == 1)

p <- ggplot(queens\_df, aes(rows, cols)) +

geom\_tile(data=board\_df, aes(fill=check), colour='black') +

geom\_text(label='\u2655', family="Arial Unicode MS", size = 8) +

theme\_void() +

coord\_equal() +

scale\_fill\_manual(values = c('TRUE'='white', 'FALSE'='grey70')) +

theme(

legend.position = 'none'

)

if (is.null(title)) {

p <- p + labs(title = paste("Queens:", deparse(as.numeric(queens))))

} else {

p <- p + labs(title = title)

}

}

#-----------------------------------------------------------------------------

# Start with no queens placed and generate all solutions.

#-----------------------------------------------------------------------------

solutions <- place\_queen()

v=1:8

f=function(q){L=length(q)

if(L==8){q}else{flatten(map(setdiff(v,c(q,q+L:1,q-L:1)),~f(c(q,.))))}}

s=data.frame(c=unlist(f(c())),r=v,x=rep(1:92,e=8),z=3)

b=mutate(crossing(c=v,r=v),z=(c+r)%%2)

g=geom\_tile

ggplot(s,aes(r,c,fill=z))+g(d=b)+g()+facet\_wrap(~x)

**5.2 Python**

"""The n queens puzzle."""

class NQueens:

"""Generate all valid solutions for the n queens puzzle"""

def \_\_init\_\_(self, size):

# Store the puzzle (problem) size and the number of valid solutions

self.size = size

self.solutions = 0

self.solve()

def solve(self):

"""Solve the n queens puzzle and print the number of solutions"""

positions = [-1] \* self.size

self.put\_queen(positions, 0)

print("Found", self.solutions, "solutions.")

def put\_queen(self, positions, target\_row):

"""

Try to place a queen on target\_row by checking all N possible cases.

If a valid place is found the function calls itself trying to place a queen

on the next row until all N queens are placed on the NxN board.

"""

# Base (stop) case - all N rows are occupied

if target\_row == self.size:

self.show\_full\_board(positions)

# self.show\_short\_board(positions)

self.solutions += 1

else:

# For all N columns positions try to place a queen

for column in range(self.size):

# Reject all invalid positions

if self.check\_place(positions, target\_row, column):

positions[target\_row] = column

self.put\_queen(positions, target\_row + 1)

def check\_place(self, positions, ocuppied\_rows, column):

"""

Check if a given position is under attack from any of

the previously placed queens (check column and diagonal positions)

"""

for i in range(ocuppied\_rows):

if positions[i] == column or \

positions[i] - i == column - ocuppied\_rows or \

positions[i] + i == column + ocuppied\_rows:

return False

return True

def show\_full\_board(self, positions):

"""Show the full NxN board"""

for row in range(self.size):

line = ""

for column in range(self.size):

if positions[row] == column:

line += "Q "

else:

line += ". "

print(line)

print("\n")

def show\_short\_board(self, positions):

"""

Show the queens positions on the board in compressed form,

each number represent the occupied column position in the corresponding row.

"""

line = ""

for i in range(self.size):

line += str(positions[i]) + " "

print(line)

def main():

"""Initialize and solve the n queens puzzle"""

NQueens(8)

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

# execute only if run as a script

main()

**VI. Danh sách liên kết đơn**

**6.1 R**

lst <- list() # creates an empty (length zero) list

lst[[1]] <- 1 # automagically extends the lst

lst[[2]] <- 2 # ditto

lst

**6.2 Python**

class Node:

# Function to initialise the node object

def \_\_init\_\_(self, data):

self.data = data # Assign data

self.next = None # Initialize next as null

# Linked List class contains a Node object

class LinkedList:

# Function to initialize head

def \_\_init\_\_(self):

self.head = None

# Code execution starts here

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

# Start with the empty list

llist = LinkedList()

llist.head = Node(1)

second = Node(2)

third = Node(3)

llist.head.next = second; # Link first node with second

second.next = third; # Link second node with the third node

**VII. Danh sách liên kết kép**

**7.1 R**

lst <- list(1, 2, 3, 4, 5) # a list of 5 items

lst <- vector("list", 10000) # 10000 NULLs

lst[[1]] <- 1

lst[[10000]] <- 10000 # lst now contains 1, NULL, ..., NULL, 10000

lst

**7.2 Python**

class Node:

def \_\_init\_\_(self, next=None, prev=None, data=None):

self.next = next # reference to next node in DLL

self.prev = prev # reference to previous node in DLL

self.data = data

# Adding a node at the front of the list

def push(self, new\_data):

# 1 & 2: Allocate the Node & Put in the data

new\_node = Node(data = new\_data)

# 3. Make next of new node as head and previous as NULL

new\_node.next = self.head

new\_node.prev = None

# 4. change prev of head node to new node

if self.head is not None:

self.head.prev = new\_node

# 5. move the head to point to the new node

self.head = new\_node

**VIII. Cài đặt ngăn xếp**

**8.1 R**

# Stack

s <- stack()

for (i in 1:3) push(s, i)

str(s)

pop(s)

str(s)

pop(s)

str(s)

pop(s)

str(s)

**8.2 Python**

from sys import maxsize

# Function to create a stack. It initializes size of stack as 0

def createStack():

stack = []

return stack

# Stack is empty when stack size is 0

def isEmpty(stack):

return len(stack) == 0

# Function to add an item to stack. It increases size by 1

def push(stack, item):

stack.append(item)

print(item + " pushed to stack ")

# Function to remove an item from stack. It decreases size by 1

def pop(stack):

if (isEmpty(stack)):

return str(-maxsize -1) # return minus infinite

return stack.pop()

# Function to return the top from stack without removing it

def peek(stack):

if (isEmpty(stack)):

return str(-maxsize -1) # return minus infinite

return stack[len(stack) - 1]

# Driver program to test above functions

stack = createStack()

push(stack, str(10))

push(stack, str(20))

push(stack, str(30))

print(pop(stack) + " popped from stack")

**IV. Cài đặt hàng đợi**

**9.1 R**

install.packages('dequer')

library(dequer)

# Queue

q <- queue()

for (i in 1:3) pushback(q, i)

str(q)

## queue of 3

## $ : int 1

## $ : int 2

## $ : int 3

pop(q)

## [1] 1

str(q)

## queue of 2

## $ : int 2

## $ : int 3

pop(q)

## [1] 2

str(q)

## queue of 1

## $ : int 3

pop(q)

## [1] 3

str(q)

**9.2 Python**

class Queue:

def \_\_init\_\_(self):

self.s1 = []

self.s2 = []

def enQueue(self, x):

# Move all elements from s1 to s2

while len(self.s1) != 0:

self.s2.append(self.s1[-1])

self.s1.pop()

# Push item into self.s1

self.s1.append(x)

# Push everything back to s1

while len(self.s2) != 0:

self.s1.append(self.s2[-1])

self.s2.pop()

# Dequeue an item from the queue

def deQueue(self):

# if first stack is empty

if len(self.s1) == 0:

print("Q is Empty")

# Return top of self.s1

x = self.s1[-1]

self.s1.pop()

return x

# Driver code

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

q = Queue()

q.enQueue(1)

q.enQueue(2)

q.enQueue(3)

print(q.deQueue())

print(q.deQueue())

print(q.deQueue())

**X. Duyệt cây theo thứ tự trước**

**10.1 R**

install.packages('data.tree')

library(data.tree)

# thiet lap cay

acme <- Node$new("A")

accounting <- acme$AddChild("B")

software <- accounting$AddChild("C")

standards <- accounting$AddChild("D")

research <- acme$AddChild("E")

newProductLine <- research$AddChild("F")

newLabs <- research$AddChild("G")

it <- acme$AddChild("H")

outsource <- it$AddChild("I")

agile <- it$AddChild("J")

goToR <- it$AddChild("K")

print(acme)

# Duyet cay Truoc

acme$Get('level')

**10.2 Python**

class Node:

def \_\_init\_\_(self, data):

self.left = None

self.right = None

self.data = data

# Insert Node

def insert(self, data):

if self.data:

if data < self.data:

if self.left is None:

self.left = Node(data)

else:

self.left.insert(data)

elif data > self.data:

if self.right is None:

self.right = Node(data)

else:

self.right.insert(data)

else:

self.data = data

# Print the Tree

def PrintTree(self):

if self.left:

self.left.PrintTree()

print( self.data),

if self.right:

self.right.PrintTree()

# Inorder traversal

# Left -> Root -> Right

def inorderTraversal(self, root):

res = []

if root:

res = self.inorderTraversal(root.left)

res.append(root.data)

res = res + self.inorderTraversal(root.right)

return res

root = Node(27)

root.insert(14)

root.insert(35)

root.insert(10)

root.insert(19)

root.insert(31)

root.insert(42)

print(root.inorderTraversal(root))

**XI. Duyệt cây theo thứ tự sau**

**11. R**  
install.packages('data.tree')

library(data.tree)

# thiet lap cay

acme <- Node$new("A")

accounting <- acme$AddChild("B")

software <- accounting$AddChild("C")

standards <- accounting$AddChild("D")

research <- acme$AddChild("E")

newProductLine <- research$AddChild("F")

newLabs <- research$AddChild("G")

it <- acme$AddChild("H")

outsource <- it$AddChild("I")

agile <- it$AddChild("J")

goToR <- it$AddChild("K")

print(acme)

# Duyet Cay sau

acme$Get('level', traversal = "post-order")

**11.2 Python**

class Node:

def \_\_init\_\_(self, data):

self.left = None

self.right = None

self.data = data

# Insert Node

def insert(self, data):

if self.data:

if data < self.data:

if self.left is None:

self.left = Node(data)

else:

self.left.insert(data)

elif data > self.data:

if self.right is None:

self.right = Node(data)

else:

self.right.insert(data)

else:

self.data = data

# Print the Tree

def PrintTree(self):

if self.left:

self.left.PrintTree()

print( self.data),

if self.right:

self.right.PrintTree()

# Preorder traversal

# Root -> Left ->Right

def PreorderTraversal(self, root):

res = []

if root:

res.append(root.data)

res = res + self.PreorderTraversal(root.left)

res = res + self.PreorderTraversal(root.right)

return res

root = Node(27)

root.insert(14)

root.insert(35)

root.insert(10)

root.insert(19)

root.insert(31)

root.insert(42)

print(root.PreorderTraversal(root))

**XII. Cài đặt đồ thị vô hướng**

**12.1 install.packages('igraph')**

install.packages('igraph')

update.packages("igraph")

library(igraph)

# Vô hướng

help(graph)

vo\_huong = make\_graph( ~ A-B-C-D-F-A, E-A:B:C:D:F)

plot(vo\_huong)

**12.2 python**

import networkx as nx

import matplotlib.pyplot as plt

#Đồ thị vô hướng

g = nx.DiGraph()

g.add\_nodes\_from([1,2,3,4,5])

g.add\_edge(1,2)

g.add\_edge(4,2)

g.add\_edge(3,5)

g.add\_edge(2,3)

g.add\_edge(2,5)

g.add\_edge(1,4)

g.add\_edge(3,4)

nx.draw(g,with\_labels=True, arrows=False)

plt.draw()

plt.show()

**XIII. Cài đặt đồ thị có hướng**

**13.1 R**

install.packages('igraph')

library(igraph)

#Co huong

help(make\_directed\_graph)

a = make\_graph(c(1, 2, 2, 3, 3, 4, 5, 6, 4,1, 2,4, 4,5, 6,1), directed = TRUE)

plot(a)

**13.2 Python**

import networkx as nx

import matplotlib.pyplot as plt

# Đồ thị có hướng

g = nx.DiGraph()

g.add\_nodes\_from([1,2,3,4,5])

g.add\_edge(1,2)

g.add\_edge(4,2)

g.add\_edge(3,5)

g.add\_edge(2,3)

g.add\_edge(2,5)

g.add\_edge(1,4)

g.add\_edge(3,4)

nx.draw(g,with\_labels=True)

plt.draw()

plt.show()

**XIV. Cài đặt thuật toán sắp xếp chọn**

**14.1 R**

dulieu <- c(32,17,49,98,06,25,53,61)

sort <- function(x) {

n<- length(x)

for (i in 1:(n-1)) {

for (j in (i+1):n) {

if(x[j] < x[i]) {

temp <-x[i]

x[i] <- x[j]

x[j] <- temp

}

}

}

return(x)

}

sort(dulieu)

selfsort <- function(x) {

if (length(x)>1) {

min <- which.min(x)

c(x[min], selfsort(x[-min]))

} else x

}

selfsort(dulieu)

**14.2 Python**

def selection\_sort(nums):

for i in range(len(nums)):

lowest\_value\_index = i

for j in range(i + 1, len(nums)):

if nums[j] < nums[lowest\_value\_index]:

lowest\_value\_index = j

nums[i], nums[lowest\_value\_index] = nums[lowest\_value\_index], nums[i]

random\_list\_of\_nums = [12, 8, 3, 20, 11]

selection\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

**XV. Cài đặt thuật toán liên kết chèn**

**15.1 R**

#sap xep chen

sort1 <- function(x) {

n <- length(x)

for (i in 2:n) {

temp <- x[i]

j = i-1

while ((x[i] > temp)&&(j>0)) {

x[j+1] = x[j]

j = j-1

}

x[j+1] = temp

}

return(x)

}

dulieu <- c(5, 2, 4, 6, 1, 3)

sort1(dulieu)

**15.2 python**

def insertion\_sort(nums):

for i in range(1, len(nums)):

item\_to\_insert = nums[i]

j = i - 1

while j >= 0 and nums[j] > item\_to\_insert:

nums[j + 1] = nums[j]

j -= 1

nums[j + 1] = item\_to\_insert

random\_list\_of\_nums = [9, 1, 15, 28, 6]

insertion\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

**XVI. Cài đặt thuật toán sắp xếp nổi bọt**

**16.1**

#sap xep noi bot

bubble <- function(x){

n<-length(x)

for(j in 1:(n-1)){

for(i in 1:(n-j)){

if(x[i]>x[i+1]){

temp<-x[i]

x[i]<-x[i+1]

x[i+1]<-temp

}

}

}

return(x)

}

x <- c(5, 2, 4, 6, 1, 3)

bubble(x)

**16.2 Python**

def bubble\_sort(nums):

swapped = True

while swapped:

swapped = False

for i in range(len(nums) - 1):

if nums[i] > nums[i + 1]:

nums[i], nums[i + 1] = nums[i + 1], nums[i]

swapped = True

random\_list\_of\_nums = [25, 2, 45, 1, 34]

bubble\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

**XVII. Cài đặt thuật toán sắp xếp nhanh**

**17.1 R**

# quick sort

quick\_sort<-function(x){

if(length(x)<=1) return(x)

pivot<-x[1]

rest<-x[-1]

pivot\_less<-quick\_sort(rest[rest<pivot])

pivot\_greater<-quick\_sort(rest[rest>=pivot])

return(c(pivot\_less,pivot,pivot\_greater))

}

dulieu = c ( 32 , 17 , 49 , 98 , 06 , 25 , 53 , 61 )

quickSort(dulieu)

**17.2 Python**

def partition(nums, low, high):

pivot = nums[(low + high) // 2]

i = low - 1

j = high + 1

while True:

i += 1

while nums[i] < pivot:

i += 1

j -= 1

while nums[j] > pivot:

j -= 1

if i >= j:

return j

nums[i], nums[j] = nums[j], nums[i]

def quick\_sort(nums):

def \_quick\_sort(items, low, high):

if low < high:

split\_index = partition(items, low, high)

\_quick\_sort(items, low, split\_index)

\_quick\_sort(items, split\_index + 1, high)

\_quick\_sort(nums, 0, len(nums) - 1)

random\_list\_of\_nums = [22, 5, 1, 18, 99]

quick\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

**XVIII Cài đặt thuật toán Heapsort**

**18.1 R**

heapify <- function(array, n, i)

{

parent <- i

leftChild <- 2 \* (i - 1) + 1

rightChild <- 2 \* (i - 1) + 2

if ((leftChild < n) & (array[parent] < array[leftChild]))

{

parent <- leftChild

}

if ((rightChild < n) & (array[parent] < array[rightChild]))

{

parent <- rightChild

}

if (parent != i) {

array <- replace(array, c(i, parent), array[c(parent, i)])

array <- heapify(array, n, parent)

}

array

}

heapSort <- function(array)

{

n <- length(array)

for (i in floor(n / 2):1) {

array <- heapify(array, n, i)

}

for (i in n:1) {

array <- replace(array, c(i, 1), array[c(1, i)])

array <- heapify(array, i, 1)

}

array

}

array <- c(32, 17, 49, 98, 6, 25, 53, 61)

heapSort(array)

**18.2 Python**

def heapify(nums, heap\_size, root\_index):

largest = root\_index

left\_child = (2 \* root\_index) + 1

right\_child = (2 \* root\_index) + 2

if left\_child < heap\_size and nums[left\_child] > nums[largest]:

largest = left\_child

if right\_child < heap\_size and nums[right\_child] > nums[largest]:

largest = right\_child

if largest != root\_index:

nums[root\_index], nums[largest] = nums[largest], nums[root\_index]

heapify(nums, heap\_size, largest)

def heap\_sort(nums):

n = len(nums)

for i in range(n, -1, -1):

heapify(nums, n, i)

for i in range(n - 1, 0, -1):

nums[i], nums[0] = nums[0], nums[i]

heapify(nums, i, 0)

random\_list\_of\_nums = [35, 12, 43, 8, 51]

heap\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

**XIX Cài đặt thuật toán sắp xếp trộn**

**19.1 R**  
mmerge<-function(a,b) {

r<-numeric(length(a)+length(b))

ai<-1; bi<-1; j<-1;

for(j in 1:length(r)) {

if((ai<=length(a) && a[ai]<b[bi]) || bi>length(b)) {

r[j] <- a[ai]

ai <- ai+1

} else {

r[j] <- b[bi]

bi <- bi+1

}

}

r

}

mmergesort<-function(A) {

if(length(A)>1) {

q <- ceiling(length(A)/2)

a <- mmergesort(A[1:q])

b <- mmergesort(A[(q+1):length(A)])

mmerge(a,b)

} else {

A

}

}

x<-c(18, 16, 8, 7, 6, 3, 11, 9, 15, 1)

mmergesort(x)

**19.2 Python**

def merge(left\_list, right\_list):

sorted\_list = []

left\_list\_index = right\_list\_index = 0

left\_list\_length, right\_list\_length = len(left\_list), len(right\_list)

for \_ in range(left\_list\_length + right\_list\_length):

if left\_list\_index < left\_list\_length and right\_list\_index < right\_list\_length:

if left\_list[left\_list\_index] <= right\_list[right\_list\_index]:

sorted\_list.append(left\_list[left\_list\_index])

left\_list\_index += 1

else:

sorted\_list.append(right\_list[right\_list\_index])

right\_list\_index += 1

elif left\_list\_index == left\_list\_length:

sorted\_list.append(right\_list[right\_list\_index])

right\_list\_index += 1

elif right\_list\_index == right\_list\_length:

sorted\_list.append(left\_list[left\_list\_index])

left\_list\_index += 1

return sorted\_list

def merge\_sort(nums):

if len(nums) <= 1:

return nums

mid = len(nums) // 2

left\_list = merge\_sort(nums[:mid])

right\_list = merge\_sort(nums[mid:])

return merge(left\_list, right\_list)

random\_list\_of\_nums = [120, 45, 68, 250, 176]

random\_list\_of\_nums = merge\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)