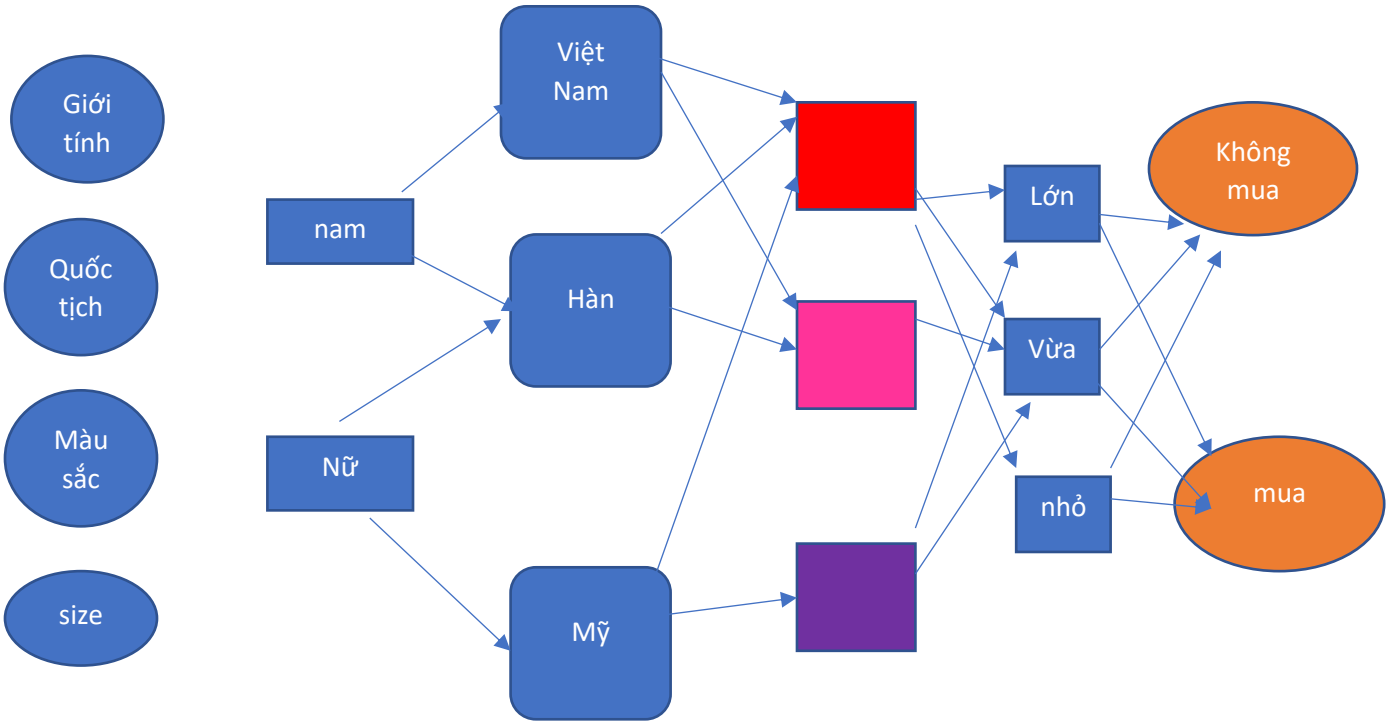
Câu 1:

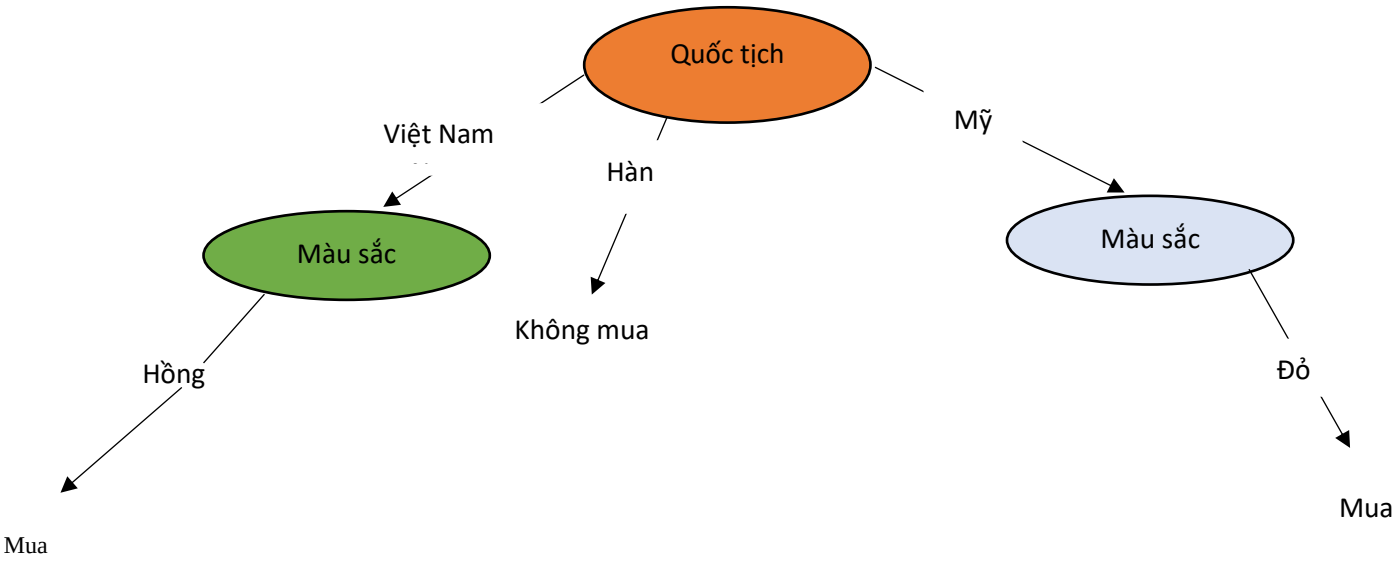
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000
int a[MAX],n;


void Doc(){


	FILE *f = fopen("C:\\Users\\DELL\\OneDrive\\Desktop\\THUC TAP CO SO\\SAP XEP\\QuickSort\\FILE.dat", "r");
   fscanf(f, "%d", &n);
	for (int i = 0; i < n; i++)
	{
				fscanf(f, "%d", &a[i]);
	}
		printf("\nChieu dai mang la: %d",n);
	printf("\nDanh sach cac phan tu mang:\t");
   for (int i=0; i<n; ++i)
	{
		printf("%d  ",a[i]);
	}
```

```
    printf("\n");
    fclose(f);
}
void Swap(int &a, int &b)
{
        int t=a; a=b; b=t;
}
void QuickSort(int a[], int l, int r)
{       int i=l; int j=r; int mid= a[(l+r)/2];
        do
        {
                while(a[i]<mid)
                        i++;
                while (mid<a[j])
                        j--;
                if(i<=j)
                {
                        Swap(a[i],a[j]);
                        i++; j--;
                }
                } while (i<=j);
                if(i<r)
                        QuickSort(a,i,r);
                if(l<j)
                        QuickSort(a,l,j);
```

```
}
void Ghi()
{
	FILE *fp;
	fp=fopen("C:\\Users\\DELL\\OneDrive\\Desktop\\THUC TAP CO SO\\SAP XEP\\QuickSort\\KQ.dat","w");
	for(int i=1; i<n;i++)
	{
		fprintf(fp,"%d\n",a[i]);
	}



    fclose(fp);
}

int main()
{
	Doc();
	QuickSort(a,0,n-1);
	Ghi();
}
```


Giới tính
Quốc tịch
Màu sắc
size
nam
Nữ
Việt Nam
Hàn
Mỹ
Lớn
Vừa
nhỏ
Không mua
mua

Câu 2b.


Quốc tịch
Việt Nam
Hàn
Mỹ
Màu sắc
Không mua
Màu sắc
Hồng
Đỏ
Mua
Mua

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1],4)
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(self.y.shape)
    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))
    def backprop(self):
        # application of the chain rule to find derivative of the loss function with respect to weights2 and weights1
        d_weights2 = np.dot(self.layer1.T, (2*(self.y - self.output) * sigmoid_derivative(self.output)))
        d_weights1 = np.dot(self.input.T,  (np.dot(2*(self.y - self.output) * sigmoid_derivative(self.output), self.weights2.T) * sigmoid_derivative(self.layer1)))
        # update the weights with the derivative (slope) of the loss function
        self.weights1 += d_weights1
        self.weights2 += d_weights2
from future import print_function
                import cv2, re, numpy as np, random, math, time, thread, decimal, tkMessageBox, tkSimpleDialog, timeit
```

```
class neural_network:


    # Construct object to develop specific network structure
    def initilize_nn(self, hidden_layers, input_count, output_count,
matrix_data,
                     matrix_targets, biases_for_non_input_layers,
                     learning_constant, testing_mode, weight_range, epochs,
                     data_to_test, dataset_meta, data_total, has_alphas,
                     user_interface):


      self.user_interface = user_interface
      if not self.user_interface.cancel_training:
       # Set all values from request if not cancelled
       self.user_interface.print_console(
          "\n\n\n---------------------------- \n Constructing neural network
\n\n"
       )
       self.all_weights = []
       self.nn_neurons = []
       self.biases_weights = []
       self.epochs = epochs
       divider_to_test = float(data_to_test) / 100.0
       self.test_data_amount = int(round(divider_to_test * data_total))
```

```
    self.dataset_meta = dataset_meta
    self.has_alphas = has_alphas
    self.matrix_data = matrix_data
    self.hidden_layers = hidden_layers
    self.matrix_targets = matrix_targets
    self.learning_constant = learning_constant
    self.output_count = output_count
    self.input_count = input_count
    self.testing_mode = testing_mode
    self.biases_for_non_input_layers = biases_for_non_input_layers
    self.weight_range = weight_range
    self.success_records = []
    self.is_small_data = len(self.matrix_targets) <= 1000


    self.populate_nn_neurons()
    self.populate_all_weights()


# Design neuron structure based on requested amounts
def populate_nn_neurons(self):
  nn_inputs = np.zeros(self.input_count)
  nn_outputs = np.zeros(self.output_count)  # Start will zero values
  self.nn_neurons.append(nn_inputs)
  for i in self.hidden_layers:
```

```
    hidden_layer = np.zeros(i)
    self.nn_neurons.append(hidden_layer)
  self.nn_neurons.append(nn_outputs)


def populate_all_weights(self):
  for neuron_layer in range(1, len(
      self.nn_neurons)):  # For all neuron layers, process weight values
    layer_length = len(self.nn_neurons[neuron_layer])
    weight_layer = []
    for single_neuron in range(0, layer_length):
      prev_layer_count = len(self.nn_neurons[neuron_layer - 1])
      neuron_weights = self.initilize_weights(
          prev_layer_count)  # Produce weight values for parent neuron
      weights_change_record_neuron = np.zeros(prev_layer_count)
      weight_layer.append(neuron_weights)
    self.all_weights.append(weight_layer)
  # Do the same for bias weights
  for layer_count in range(0, len(self.biases_for_non_input_layers)):
    single_bias_weights = []
    single_bias_weights_change = []
    if (self.biases_for_non_input_layers[layer_count] != 0):
      bias_input_count = len(self.nn_neurons[layer_count + 1])
      single_bias_weights = self.initilize_weights(bias_input_count)
      single_bias_weights_change = np.zeros(bias_input_count)
```

```
                self.biases_weights.append(single_bias_weights)
            def initilize_weights(
                self, size):  # Get weight values as random values within bounds
              if (len(self.weight_range) == 1):
                upper_bound = self.weight_range[0]
                lower_bound = upper_bound
              else:
                upper_bound = self.weight_range[1]
                lower_bound = self.weight_range[0]
              return np.random.uniform(low=lower_bound, high=upper_bound,
size=(size))
            def feed_forward(self, matrix):
              self.populate_input_layer(matrix)  # Send single data row to
network
              for after_input_layer in range(1, len(self.nn_neurons)):
                hidden_neuron_sums = np.dot(
                    np.asarray(self.all_weights[after_input_layer - 1]),
                    self.nn_neurons[after_input_layer - 1])
                if len(self.biases_weights[after_input_layer - 1]) > 0:
                  bias_vals = (self.biases_for_non_input_layers[after_input_layer -
1] *
                               self.biases_weights[after_input_layer - 1])
                  hidden_neuron_sums += bias_vals
                self.nn_neurons[after_input_layer] = self.activate_threshold(
                    hidden_neuron_sums, "sigmoid")
            def populate_input_layer(
```

```
                self, data):  # Put data row on to input layer ready for feed forward
               if (self.has_alphas):
                encoded_input = []
                item_i = 0
                for item_pos in self.dataset_meta["alphas"]:
                 if (int(item_pos) not in self.dataset_meta["target_info"][2]
                  ):  #If the value is not a target value, add to input
                   # Process each bit of data, and construct vector if values are
classified
                   bin_vec =
self.user_interface.data_processor.alpha_class_to_binary_vector(
                     data[item_i], self.dataset_meta["alphas"][item_pos])
                   encoded_input += bin_vec
                   item_i += 1


               else:
                encoded_input = data
               self.nn_neurons[0] = encoded_input


              testing_output_mode = False
              test_counter = 0
              correct_count = 0
              error_by_1000 = 0
              error_by_1000_counter = 1
```

```
                output_error_total = 0



                def construct_target_for_bp(self, target_val):
                  # Construct binary vector if numeric classification
                  if (self.dataset_meta["target_info"][0] == "Binary" and
                      str(target_val[0]).isdigit()):
                    target_vector =
self.user_interface.data_processor.populate_binary_vector(
                        target_val[0], self.output_count)
                  else:
                    # Construct binary vector if alpha classification
                    target_vector = []
                    t_i = 0
                    for t_val in target_val:
                      t_pos = self.dataset_meta["target_info"][2][t_i]
                      bin_vec =
self.user_interface.data_processor.alpha_class_to_binary_vector(
                          t_val, self.dataset_meta["alphas"][t_pos])
                      target_vector += bin_vec
                      t_i += 1
                  return target_vector



                def back_propagate(self, target_val, repeat_count):
```

```
                # Ready target values to be compared to output in conforming
structure
                target_vector = self.construct_target_for_bp(target_val)



                # Determine how success must be judged
                if (len(self.nn_neurons[-1]) > 1):
                  outputs_as_list = self.nn_neurons[-1].tolist()
                  # Judge by one-hot encoding output value being index of highest
target value
                  success_condition = (outputs_as_list.index(
                      max(outputs_as_list)) ==
target_vector.index(max(target_vector)))
                else:
                  # Judge by accuracy of real value
                  success_condition = (round(self.nn_neurons[-1][0]) ==
target_vector)



                # Measure/track success for graphs
                if (self.test_counter >= len(self.matrix_data) -
self.test_data_amount):
                  if success_condition:
                    self.correct_count += 1
                if not success_condition:
                  self.error_by_1000 += 1
```

```
                if self.error_by_1000_counter % 1000 == 0:
                    # Feed error data to graph
                    self.user_interface.animate_graph_figures(0, self.error_by_1000 /
10)
                    self.error_by_1000 = 0
                    self.error_by_1000_counter = 0


                # The backpropagation. Start at output layer, and work backwards...
                for weight_layer_count in range(len(self.all_weights) - 1, -1, -1):


                    # Get neuron values of given layer, and add dimension for
conforming with activated_to_sum_step
                    weight_neuron_vals = np.expand_dims(
                        self.nn_neurons[weight_layer_count + 1], axis=1)
                    target_vector = np.expand_dims(target_vector, axis=1)


                    activated_to_sum_step = weight_neuron_vals * (1 -
weight_neuron_vals)


                    # If output layer (first step of BP), compare to target value
```

```
                if (weight_layer_count == len(self.all_weights) - 1):
                  back_prop_cost_to_sum = (
                      weight_neuron_vals - target_vector) * activated_to_sum_step
                else:  # Otherwise, compare to previous propagated layer values
                  trans_prev_weights = np.asarray(
                      self.all_weights[weight_layer_count + 1]).transpose()
                  back_prop_cost_to_sum = np.dot(
                      trans_prev_weights, back_prop_cost_to_sum) *
activated_to_sum_step



                # If biases being used, BP them too.
                if len(self.biases_weights[weight_layer_count]) > 0:
                  current_bias_weight_vals =
self.biases_weights[weight_layer_count]
                  final_bias_change = self.learning_constant *
back_prop_cost_to_sum.flatten(
                  )
                  self.biases_weights[
                      weight_layer_count] = current_bias_weight_vals -
final_bias_change



                # Get neuron values on layer ahead and BP to the weights
                input_neuron_vals = np.expand_dims(
                   self.nn_neurons[weight_layer_count], axis=1)
                full_back_prop_sum_to_input = np.dot(back_prop_cost_to_sum,
```

```
                                        input_neuron_vals.transpose())


                    # Update weight values using learning rate
                    current_weight_vals = self.all_weights[weight_layer_count]
                    new_weight_vals = current_weight_vals - (
                        self.learning_constant * full_back_prop_sum_to_input)
                    self.all_weights[weight_layer_count] = new_weight_vals



                  self.test_counter += 1
                  self.error_by_1000_counter += 1



                def train(self):
                  if not self.user_interface.cancel_training:
                    success_list = []
                    hidden_layer_str = ""
                    for layerc in self.hidden_layers:  # Construct a list of hidden layer
values for console history
                      hidden_layer_str += str(layerc) + ","
                    hidden_layer_str = hidden_layer_str[0:-1]
                    cancel_training = False
                    # Output main neural network hyperparameters for console history
r
                    self.user_interface.print_console(" TRAINING \n")
```

```
                self.user_interface.print_console("With learning rate: " +
                                                  str(self.learning_constant))
                self.user_interface.print_console("With hidden layers: " +
                                                  str(hidden_layer_str))
                self.user_interface.print_console("With test amount by epoch size:
" +
                                                  str(self.test_data_amount) + "/" +
                                                  str(len(self.matrix_targets)))
                self.user_interface.print_console("With epoch count: " +
str(self.epochs))



                if self.testing_mode:
                  self.repeat_count = 5000



                epoch_times = []
                # Iterate over dataset for each epoch
                for epoch in range(1, self.epochs + 1):
                  pre_epoch_time = time.time()  # Get inital time for epoch time
tracking
                  matrix_count = 0
                  for matrix in self.matrix_data:
                    if self.user_interface.cancel_training:
                      # Cancel training if requested
                      break
```

```
                target_vals = self.matrix_targets[matrix_count]
                self.feed_forward(
                    matrix)  # Send data to network and initiate the feed forward
                self.back_propagate(target_vals, epoch)  # After outputs
produced, BP.
                matrix_count += 1
            if self.user_interface.cancel_training:
                break



            success_p = (float(self.correct_count) / float(
                self.test_data_amount)) * 100  # Measure success for one
epoch



            #Send success data to UI for graph
            self.user_interface.animate_graph_figures(1, success_p)
            e_note_str = " (ep. " + str(epoch) + ")"
            success_list.append(success_p)



            #Output epoch time and latest success values on UI
            if not self.is_small_data:
                self.user_interface.update_canvas_info_label(
                    "Latest Success",
                    str(round(success_p, 2)) + "%" + e_note_str)
```

```
  self.test_counter = 0
  self.correct_count = 0
  post_epoch_time = time.time() - pre_epoch_time
  if not self.is_small_data:
    self.user_interface.update_canvas_info_label(
        "Epoch Duration",
        str(round(post_epoch_time, 2)) + "s " + e_note_str)
  epoch_times.append(post_epoch_time)


#Complete training, cancel it, output results.
if len(success_list) > 0:
  av_success = sum(success_list) / len(success_list)
  highest_success = max(success_list)
  av_epoch_time = round(sum(epoch_times) / len(epoch_times), 5)
else:
  av_success = "N/A"
  highest_success = "N/A"
  av_epoch_time = "N/A"
training_done_msg = "FINISHED"
if self.user_interface.cancel_training:
  training_done_msg = "CANCELLED"
else:
```

```
                self.user_interface.cancel_learning()
                self.user_interface.print_console(training_done_msg)
                self.user_interface.print_console("AVERAGE SUCCESS: " +
str(av_success) +
                                                  "%")
                self.user_interface.print_console("HIGHEST SUCCESS: " +
                                                  str(highest_success) + "%")
                self.user_interface.print_console("TOTAL TIME: " +
str(sum(epoch_times)) +
                                                  "s")
                self.user_interface.print_console("AVERAGE EPOCH TIME: " +
                                                  str(av_epoch_time) + "s")



            def activate_threshold(self, value, type):
                if (type == "step"):
                    if (value >= 0.5):
                        return 1
                    else:
                        return 0
                elif (type == "sigmoid"):
                    return 1 / (1 + np.exp(-value))
```

câu 3

```
import os
import playsound
import speech_recognition as sr
import time
import sys
```

```
import ctypes
import wikipedia
import datetime
import json
import re
import webbrowser
import smtplib
import requests
import urllib
import urllib.request as urllib2
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from webdriver_manager.chrome import ChromeDriverManager
from time import strftime
from gtts import gTTS
from youtube_search import YoutubeSearch

wikipedia.set_lang('vi')
language = 'vi'
path = ChromeDriverManager().install()

def speak(text):
    print("Bot: {}".format(text))
    tts = gTTS(text=text, lang=language, slow=False)
    tts.save("sound.mp3")
    playsound.playsound("sound.mp3", False)
    os.remove("sound.mp3")

def get_audio():
    print("\nBot: \tĐang nghe \t --__-- \n")
    r = sr.Recognizer()
    with sr.Microphone() as source:
        print("Tôi: ", end='')
        audio = r.listen(source, phrase_time_limit=8)
        try:
            text = r.recognize_google(audio, language="vi-VN")
            print(text)
            return text.lower()
        except:
            print("...")
            return 0

def stop():
    speak("Hẹn gặp lại bạn sau!")
    time.sleep(2)
```

```
def get_text():
    for i in range(3):
        text = get_audio()
        if text:
            return text.lower()
        elif i < 2:
            speak("Máy không nghe rõ. Bạn nói lại được không!")
            time.sleep(3)
    time.sleep(2)
    stop()
    return 0

def hello(name):
    day_time = int(strftime('%H'))
    if day_time < 12:
        speak("Chào buổi sáng bạn {}. Chúc bạn một ngày tốt lành.".format(name))
    elif 12 <= day_time < 18:
        speak("Chào buổi chiều bạn {}. Bạn đã dự định gì cho chiều nay chưa.".format(name))
    else:
        speak("Chào buổi tối bạn {}. Bạn đã ăn tối chưa nhỉ.".format(name))
    time.sleep(5)

def get_time(text):
    now = datetime.datetime.now()
    if "giờ" in text:
        speak('Bây giờ là %d giờ %d phút %d giây' % (now.hour, now.minute, now.second))
    elif "ngày" in text:
        speak("Hôm nay là ngày %d tháng %d năm %d" %
              (now.day, now.month, now.year))
    else:
        speak("Bot chưa hiểu ý của bạn. Bạn nói lại được không?")
    time.sleep(4)

def open_application(text):
    if "google" in text:
        speak("Mở Google Chrome")
        time.sleep(2)
        os.startfile('Desktop\\Google Chrome')
    elif "word" in text:
        speak("Mở Microsoft Word")
        time.sleep(2)
        os.startfile('Desktop\\Google Chrome')
    elif "excel" in text:
        speak("Mở Microsoft Excel")
```

```
        time.sleep(2)
        os.startfile('Desktop\\Google Chrome')
    else:
        speak("Ứng dụng chưa được cài đặt. Bạn hãy thử lại!")
        time.sleep(3)

def open_website(text):
    reg_ex = re.search('mở (.+)', text)
    if reg_ex:
        domain = reg_ex.group(1)
        url = 'https://www.' + domain
        webbrowser.open(url)
        speak("Trang web bạn yêu cầu đã được mở.")
        time.sleep(3)
        return True
    else:
        return False

def open_google_and_search(text):
    search_for = text.split("kiếm", 1)[1]
    speak('Okay!')
    driver = webdriver.Chrome(path)
    driver.get("http://www.google.com")
    que = driver.find_element_by_xpath("//input[@name='q']")
    que.send_keys(str(search_for))
    que.send_keys(Keys.RETURN)
    time.sleep(10)

def send_email(text):
    speak('Bạn gửi email cho ai nhỉ')
    time.sleep(2)
    recipient = get_text()
    if 'yến' in recipient:
        speak('Nội dung bạn muốn gửi là gì')
        time.sleep(3)
        content = get_text()
        speak('Email của bạn vùa được gửi. Bạn check lại email nhé hihi.')
        time.sleep(4)
    else:
        speak('Bot không hiểu bạn muốn gửi email cho ai. Bạn nói lại được không?')
        time.sleep(5)

def current_weather():
    speak("Bạn muốn xem thời tiết ở đâu ạ.")
    time.sleep(3)
```

```
    ow_url = "http://api.openweathermap.org/data/2.5/weather?"
    city = get_text()
    if not city:
        pass
    api_key = "fe8d8c65cf345889139d8e545f57819a"
    call_url = ow_url + "appid=" + api_key + "&q=" + city + "&units=metric"
    response = requests.get(call_url)
    data = response.json()
    if data["cod"] != "404":
        city_res = data["main"]
        current_temperature = city_res["temp"]
        current_pressure = city_res["pressure"]
        current_humidity = city_res["humidity"]
        suntime = data["sys"]
        sunrise = datetime.datetime.fromtimestamp(suntime["sunrise"])
        sunset = datetime.datetime.fromtimestamp(suntime["sunset"])
        wthr = data["weather"]
        weather_description = wthr[0]["description"]
        now = datetime.datetime.now()
        content = """
        Hôm nay là ngày {day} tháng {month} năm {year}
        Mặt trời mọc vào {hourrise} giờ {minrise} phút
        Mặt trời lặn vào {hourset} giờ {minset} phút
        Nhiệt độ trung bình là {temp} độ C
        Áp suất không khí là {pressure} héc tơ Pascal
        Độ ẩm là {humidity}%
        Trời hôm nay quang mây. Dự báo mưa rải rác ở một số nơi.""".format(day = now.day,mont
h = now.month, year= now.year, hourrise = sunrise.hour, minrise = sunrise.minute,
                                                    hourset = sunset.hour, minset = sunset.minute,
                                                    temp = current_temperature, pressure = current_pr
essure, humidity = current_humidity)
        speak(content)
        time.sleep(28)
    else:
        speak("Không tìm thấy địa chỉ của bạn")
        time.sleep(2)

def play_song():
    speak('Xin mời bạn chọn tên bài hát')
    time.sleep(2)
    mysong = get_text()
    while True:
        result = YoutubeSearch(mysong, max_results=10).to_dict()
        if result:
            break
```

```
    url = 'https://www.youtube.com' + result[0]['url_suffix']
    webbrowser.open(url)
    speak("Bài hát bạn yêu cầu đã được mở.")
    time.sleep(3)

def change_wallpaper():
    api_key = 'RF3LyUUIyogjCpQwlf-zjzCf1JdvRwb--SLV6iCzOxw'
    url = 'https://api.unsplash.com/photos/random?client_id=' + \
        api_key  # pic from unspalsh.com
    f = urllib2.urlopen(url)
    json_string = f.read()
    f.close()
    parsed_json = json.loads(json_string)
    photo = parsed_json['urls']['full']
    # Location where we download the image to.
    urllib2.urlretrieve(photo, "D:\\Download____CocCoc\\a.png")
    image=os.path.join("D:\\Download____CocCoc\\a.png")
    ctypes.windll.user32.SystemParametersInfoW(20,0,image,3)
    speak('Hình nền máy tính vừa được thay đổi')
    time.sleep(3)

def tell_me_about():
    try:
        speak("Bạn muốn nghe về gì ạ")
        time.sleep(2)
        text = get_text()
        contents = wikipedia.summary(text).split('\n')
        speak(contents[0].split(".")[0])
        time.sleep(10)
        for content in contents[1:]:
            speak("Bạn muốn nghe thêm không")
            time.sleep(2)
            ans = get_text()
            if "có" not in ans:
                break
            speak(content)
            time.sleep(10)

        speak('Cảm ơn bạn đã lắng nghe!!!')
        time.sleep(3)
    except:
        speak("Bot không định nghĩa được thuật ngữ của bạn. Xin mời bạn nói lại")
        time.sleep(5)

def help_me():
```

```
    speak("""Bot có thể giúp bạn thực hiện các câu lệnh sau đây:
    1. Chào hỏi
    2. Hiển thị giờ
    3. Mở website, application
    4. Tìm kiếm trên Google
    5. Gửi email
    6. Dự báo thời tiết
    7. Mở video nhạc
    8. Thay đổi hình nền máy tính
    9. Đọc báo hôm nay
    10. Kể bạn biết về thế giới """)
    time.sleep(27)

def read_news():
    speak("Chức năng còn đang xây dựng. Vui lòng chọn chức năng khác")
    time.sleep(5)

def assistant():
    speak("Xin chào, bạn tên là gì nhỉ?")
    time.sleep(2)
    name = get_text()
    if name:
        speak("Chào bạn {}".format(name))
        speak("Bạn cần Bot Alex có thể giúp gì ạ?")
        time.sleep(3)
        while True:
            text = get_text()
            if not text:
                break
            elif "dừng" in text or "tạm biệt" in text or "chào robot" in text or "ngủ thôi" in text:
                stop()
                break
            elif "có thể làm gì" in text:
                help_me()
            elif "chào" in text:
                hello(name)
            elif "giờ" in text or "ngày" in text:
                get_time(text)
            elif 'mở google và tìm kiếm' in text:
                open_google_and_search(text)
            elif "mở " in text:
                open_website(text)
            elif "ứng dụng" in text:
                speak("Tên ứng dụng bạn muốn mở là ")
                time.sleep(3)
```

```
            text1 = get_text()
            open_application(text1)
        elif "email" in text or "mail" in text or "gmail" in text:
            send_email(text)
        elif "thời tiết" in text:
            current_weather()
        elif "chơi nhạc" in text:
            play_song()
        elif "hình nền" in text:
            change_wallpaper()
        elif "đọc báo" in text:
            read_news()
        elif "định nghĩa" in text:
            tell_me_about()
        else:
            speak("Bạn cần Bot giúp gì ạ?")
            time.sleep(2)

assistant()
```

Trợ lý ảo

```
import speech_recognition

import pyttsx3

#import datatime import data, datatime

robot_ear = speech_recognition.Recognizer()

robot_mouth = pyttsx3.init()

robot_brain = ""

while True: # cái này để mình và robot giao tiếp liên tục thay vì nói 1 câu chương trình đã kết thúc.

    with speech_recognition.Microphone() as mic:

        print("Robot: I'm Listening")

        audio = robot_ear.listen(mic)

    print("Robot:...")

    try:

        you = robot_ear.recognizer_google(audio)
```

```
    except:
        you = ""
    if you == "":
        robot_brain = "I can't hear you, try again"
    elif "Hello" in you: # in you này thay vì chúng ta nói Hello sẽ trả ra
    #"Hello python thì nó sẽ kiểm tra là trong câu mà bạn nói có từ Hello hay không
?
        robot_brain: "Hello Python"
    #elif "Today" in you:
    #    today = date.today()
    #    robot_brain = today.strftime("%B %d, %Y")
    #elif "Time" in you:
    #    now = datetime.today()
    #    robot_brain = now.strftime("%H hours %M minutes %S seconds")
    elif "goodbye" in you: ## đoạn này khi nói goodbye thì chương trình sẽ tắt thay
vì mở liên tục khi ở phía trên
        robot_brain = "Good Bye"
        break
    else:
        robot_brain = "I'm fine thank you and you"
print("Robot:" + robot_brain)
robot_mouth.say(robot_brain)
robot_mouth.runAndWait()
```