CS/CE 102/171: Data Structures & Algorithms
Spring 2024 - Homework 02

Department of Computer Science

Dhanani School of Science and Engineering

Habib University

Contents

Introduction

1.1 Instructions

- This homework will contribute 5% towards your final grade.
- The deadline for submission of this homework is 09:00 am on Monday, April 8, 2024.
- The homework must be submitted online via CANVAS. You are required to submit a zip file that contains all the *.py* files.
- The zip file should be named as *HW_02_aa1234.zip* where *aa1234* will be replaced with your student id.
- **Files that don't follow the appropriate naming convention will not be graded.**

1.2 Grading Rubric

Your assignment will be graded according to the rubric given in the Excel file `dsa-spring24-hw2-rubric.xlsx`

1.3 Homework Objectives

Use Divide and Conquer, Graph, and Hashtable ADTs and algorithms effectively and efficiently to solve problems.

1.4 Late submission policy

You are allowed to submit late till 9:00 am on Tuesday, April 9, 2024, with a 20% penalty. No submissions after this will be accepted.

1.5 Use of AI

Taking help from any AI-based tools such as ChatGPT is strictly prohibited and will be considered plagiarism.

1.6 Viva

Course staff may call any student for Viva to provide an explanation for their submission.

1.7 Queries

You can post your Queries on the Homework 02 Discussion on Canvas or visit course staff during their office hours.

1.8 Processing Text Files

For each question, you have to implement a `main(...)` function that takes a filename as an argument. The `main(...)` function should read the input from the file and call the required functions.

You can use the following code to read the input from a file:

```
with open('input.txt') as f:
    lines = f.readlines()

input = []
for line in lines:
    line = line.strip() # remove leading and trailing spaces
    tokens = line.split() # split the line into tokens
    input.append(tokens[0]) # add the first token to the
   input list

main(input) # call the main function
```

Questions

2.1 Divide and Conquer

2.1.1 Decrypting Coordinates

A company has decided to employ the Karatsuba algorithm (see Section 2.1.5 for details) to encrypt the coordinates of their assets. The encryption process involves generating a call tree of recursive calls of the Karatsuba algorithm and sharing only the leaf nodes of this tree. Each leaf node contains a pair of integers representing the parameters of the call to the Karatsuba algorithm.

Your task is to understand the Karatsuba algorithm and design a recursive algorithm to decrypt the actual coordinates from leaf nodes.

2.1.2 Function(s) Description

You have to provide the implementation of the following function:

- `reverse_karatsuba(data) -> tuple`: A recursive function that receives leaf nodes as a parameter in the form of a nested list and returns a tuple: the original numbers used by the Karatsuba algorithm to generate given leaf nodes.
- `main(filename) -> list[tuple[int, int]]`: It receives the input file name as a parameter and returns the list tuples as coordinates for the given trees in the file.

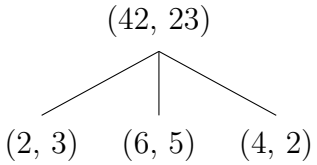
2.1.3 Input Format

The first line of the input file is an integer N where $1 \leq N \leq 100$. For the next N lines, each line consists of a nested list, with each level of the tree represented by a list. At each level of the tree, the elements can either be tuples or nested lists. If an element is a tuple, it represents a leaf node containing coordinates (x, y). If an element is a nested list, it represents a subtree of the tree, and the nested list follows the same structure as the outermost list.

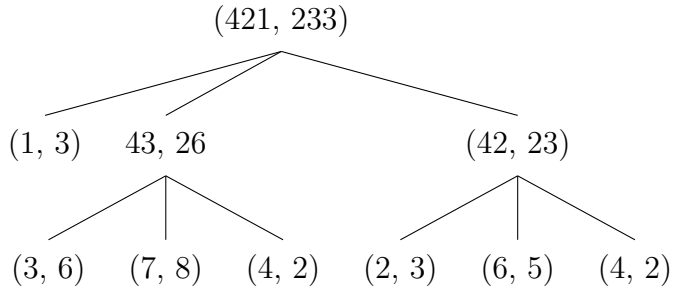
Sample Input

```
2
[(2, 3), (6, 5), (4, 2)]
[(1, 3), [(3, 6), (7, 8), (4, 2)], [(2, 3), (6, 5), (4, 2)]]
```

The following tree visualizes the calls of the Karatsuba algorithm for (42, 23). The leaf nodes of this tree are (x=2, y=3), (x=6, y=5), and (x=4, y=2). The company shares only this information. Your decryption algorithm must be able to generate the actual coordinates (42, 23) from these leaf nodes.



Similarly, the tree for the coordinates (421, 233) is represented as follows:



2.1.4 Output

The output should return a list consisting of N tuples where each tuple is a coordinate (x, y) represented as a pair of integers.

The output of the above input file should be:

```
[(42, 23), (421, 233)]
```

2.1.5 Karatsuba algorithm

The Karatsuba algorithm is a fast multiplication algorithm that efficiently multiplies two large numbers by dividing them into smaller parts, recursively solving sub-problems, and combining the results. It was discovered by Anatolii Alexeevitch Karatsuba in 1960 when he was a student.

The algorithm takes advantage of the divide-and-conquer strategy to reduce the number of arithmetic operations required for multiplication, especially for large numbers. Instead of directly multiplying two numbers digit by digit using the

traditional "long multiplication" method, the Karatsuba algorithm breaks down the numbers into smaller parts, typically of equal size, and performs intermediate multiplications to compute partial products. These partial products are then combined to obtain the final result.

The key idea behind the Karatsuba algorithm is to exploit the fact that the product of two large numbers can be expressed as a sum of smaller products of their components. By recursively applying this approach to sub-problems, the algorithm achieves a significant reduction in the number of multiplications compared to traditional methods.

The Karatsuba algorithm can be described as follows:

1. Given two numbers x and y to be multiplied, express them as follows:

$$x = x_1 \cdot B^m + x_0$$
$$y = y_1 \cdot B^m + y_0$$

   where B is the base of the number system (e.g., 10 for decimal numbers), and m is the number of digits in the smaller of the two numbers.

2. Compute the following intermediate products:

$$z_0 = x_0 \cdot y_0$$
$$z_1 = (x_1 + x_0) \cdot (y_1 + y_0)$$
$$z_2 = x_1 \cdot y_1$$

3. Compute the final result using the following formula:

$$x \cdot y = z_2 \cdot B^{2m} + (z_1 - z_2 - z_0) \cdot B^m + z_0$$

4. Recursively apply the Karatsuba algorithm to compute the intermediate products z_1 and z_2. The final result can be obtained by combining these intermediate products using the formula above.

5. The base case of the recursion occurs when the numbers are small enough to be multiplied directly using the traditional method.

6. The algorithm terminates when the base case is reached, and the final result is obtained by combining the partial products computed at each level of the recursion.

2.1.6 Pseudocode

The pseudocode for the Karatsuba algorithm can be expressed as follows:

```
    def karatsuba(x, y):
    if x < 10 or y < 10:
        return x * y

    n = max(len(str(x)), len(str(y)))
    n2 = n // 2

    high1, low1 = split(x, n2)
    high2, low2 = split(y, n2)

    z0 = karatsuba(low1, low2)
    z1 = karatsuba((low1 + high1), (low2 + high2))
    z2 = karatsuba(high1, high2)

    return (z2 * 10**(2 * n2)) + ((z1 - z2 - z0) * 10**n2) + z0

def split(num, n):
    high = num // 10**n
    low = num % 10**n
    return high, low
```

Credits: Muhammad Qasim Pasta

2.2 Hash Tables

2.2.1 Student Database

The Old School Academy is overwhelmed by paper records. Headmaster Thompson formed a team to create a Student Database, a digital record-keeping system to make student records easily accessible. If the Student Database proves its worth, The Old School Academy plans to distribute it in other schools.

You are lucky to be the part of the Student Database Development Team. Therefore, you are assigned a task to create a hash table that can read students' record from a CSV text file and create a hash table representation from it. To test this new system you will be performing some Operations, which are mentioned in the text file **Operations.txt**.

2.2.2 Function(s) Description

Following are the files which are provided to you,

- *data.csv* (Input file containing students' information)
- *Operations.txt* (Input file containing operations to be performed on hash table)
- *Hashtable.py* (Python file Containing helper functions for hash table)

- *StudentDatabase.py* (Python file for creating the hash table and performing operations on it)

You are required to implement the following functions in *StudentDatabase.py* file

- `def main(filename)` - Takes the input file (`data.csv`) as argument and returns a list of Dictionary in the following format: `[{'SNo': '1', 'ID': '88F7B33d2bcf9f5', 'FirstName': 'Shelby', 'LastName': 'Terrell', 'Sex': 'Male', 'Email': 'elijah57@example.net', 'Phone': '001-084-906-7849`
  `'Department': 'CS'}, ....]`

- `def create_studentDatabase(studentRecords)`- Takes the list of Dictionary from the main function and returns the hash table of size = 7 using the helper functions from `hashtable.py`. Hash table is a tuple of two lists, first list contains all the IDs of students and the second list contains the dictionary of students' information

- `def perform_Operations(H,operationFile)` - Takes the hash table and operations files (`Operations.txt`) as input and perform all the operations on hash table. It returns a dictionary of collision_path, the format of this dictionary is mentioned in the output section.

You are required to implement the following functions in *HashTable.py*

- `def create_hashtable(size)` - Create a hash table of input size (it will be size = 7; read the description of `def create_studentDatabase(...)`).

- `def resize_hashtable(hashtable,size,increase)` - Takes **Hashtable**, **size** and **increase** a boolean variable. The purpose of this function is to resize (increase or decrease the size of) the hash table. If the value of **increase** variable is True, the size of the hash table will increase by multiplying it by 2 and extending it to the next Prime number. For example, 7 will become 7*2=14 and the nearest next prime number to number 14 will be 17 so this will be the new size of hash table.

  If the value of **increase** variable is False, the hash table size will be reduced to half and extend it further to make it a Prime number. For example, if the size of the hash table was 17, then it will become 17//2= 8, and the nearest Prime number is 11 so it will be reduced to the size of 11.Note that the size of the hash table should not be less than 7.

- `def hash_function(key,size)` - It takes a Student ID as a key and size of the hash table. The function will return the hash value by following these steps,

  1. Sum the ASCII equivalent of each character in the key
  2. Right shift 4 bits
  3. Return the modulo size for this new value and make sure the address remains nonnegative in this process,

- `def collision_resolver(key,oldAddress,size)`- This function takes the key, OldAddress (The last address on which a collision occured) and the size of the hash table. It calculates and returns the new address using the Key Offset method using the following formula:

```
offset=Sum of Ascii Codes of key // size
address=((offset+oldAddress)%size)
```

- `def put(hashtable,key, data,size)` - The function takes hashtable, key to be inserted , the data associated with the key (a dictionary in this case) and size of the hash table as input values. This function also considers resizing/shrinking of hash table based on the load factor of **75%** and **30%**. When more than **75%** of hash table is already populated, increase the size, whereas decrease the size if it is less than **30%**. The function returns the hashtable and its size as a tuple.

- `def loadFactor(hashtable,size)` - Returns the load factor of the hashtable

- `def Update(hashtable,key, columnName, data,size,collision_path,opNumber)` It takes `hashtable`, `key` to be updated, the `ColumnName` whose value needs to be changed to `data`. It also takes a dictionary named `collision_path` and an integer `opNumber`. `opNumber` is the sequence Number of the Operation being performed and this will act as a key to the `collision_path`. To see the format of `collision_path` see the output section.

- `def get(hashtable,key,size,collision_path,opNumber)` returns the value associated with the key in hashtable or prints "Item not found"

- `def delete(hashtable, key, size,collision_path,opNumber)` - Delete key from hashtable and places a tombstone marker #

2.2.3 Input Format

There are 2 input files as mentioned earlier, the format of **input file** (`data.csv`), is given below,

```
  SNo,ID,FirstName,LastName,Sex,Email,Phone,Department
  1,88F7B33d2bcf9f5,Shelby,Terrell,Male,elijah57@example.
net,001-084-906-84,CS
  2,f90cD3E76f1A9b9,Phillip,Summers,Female,
bethany14@example.com,214.112.6044,CS
```

The first row contains the Column names for the student information. The next rows are the values associated with these columns in the same order. You are provided the information of 10 students.

Operations.txt file has the following format

```
Find bfDD7CDEF5D865B
Find 2EFC6A4e77FaEaC FirstName
Update 2EFC6A4e77FaEaC LastName Umer
Delete 2EFC6A4e77FaEaC
```

Sample operations format

- **Find bfDD7CDEF5D865B**
  This will find the ID = bfDD7CDEF5D865B from the hashtable and print the Student Info associated with this ID (in the form of Dictionary)
- **Find 2EFC6A4e77FaEaC FirstName**
  This will find the ID = 2EFC6A4e77FaEaC, from the hashtable and print the specific Column value from Student Information associated with this ID (in the form of String).
- **Update 2EFC6A4e77FaEaC LastName Smith**
  This will set the LastName of a student with ID=2EFC6A4e77FaEaC to Smith.
- **Delete 2EFC6A4e77FaEaC**
  This will delete the Student with ID= 2EFC6A4e77FaEaC.

2.2.4 Output

Besides intermediate printing of statements mentioned in the function description section, under each function the `perform_Operations()` will maintain a dictionary called `collision_path` and returns it. The purpose of this dictionary is to maintain a log of collision path for each Operation.
The dictionary should look like this:

```
{1: [15], 2: [12], 3: [13, 6], 4: [13, 6], 5: [13, 6], 6: [13, 6], 7: [13, 6], 8: [9], 9: [14], 10: [13, 5], 11: [7], 12: [5], 13: [0]}
```

Keys of this dictionary are the operation number in the order mentioned in *Operation.txt*, the values are the list of indexes on which collisions occur.

Credits: Saba Saeed

2.3 Graphs

2.3.1 Flight Network

"Fly Imaraat" is a new airline originating from Dubai, and they are flying to more than 100 cities around the world. The task is to create a graph network (using Adjacency Maps, as done in labs) for their flights destined towards different cities in North America, Central America and South America; and then use it accordingly. Some of these destinations have direct, non-stop flights originating from Dubai, whereas many others have different layovers in between. These layovers may or may not be in the specified region.

2.3.2 Function(s) Description

You have to implement the following functions in the file `FlightNetwork.py`

(a) `def create_flight_network(filename, option)` takes two parameters: `filename` which is the path of the CSV file of the dataset, and `option` as an integer:

- 1 for creating graph with flight duration as the edge weights
- 2 for creating graph with flight distance as the edge weights.

The function then returns the respective graph at the end

(b) `def get_flight_connections(graph, city, option)` takes three parameters: `graph` which is the graph created in part (a), `city` which is a string representing the vertex, and `option` which is a character: "i" for inbound flights, and "o" for outbound flights. The function then finds all the connected vertices based on the option selected, and returns the connected airport cities in the form of a list. If there is no connections to and from this city, the result will be an empty list.

(c) `def get_number_of_flight_connections(graph, city, option)` takes three parameters: `graph` which is the graph created in part (a), `city` which is a string representing the vertex, and `option` which is a character: "i" for inbound flights, and "o" for outbound flights. The function then calculates the total number of flights depending on the option selected, and returns that number.

(d) `def get_flight_details(graph, origin, destination)` takes three parameters: `graph` which is the graph created in part (a), `origin` which is a string representing the origin city, and `destination` – also a string representing the destination city. The function returns the time or distance to go from origin city to the destination city, depending on the graph being used. If the origin city is not in the graph, it returns None. If the destination is not connected to the origin city, it returns -1

(e) `def add_flight(graph, origin, destination, weight)` takes four parameters: `graph` which is the graph created in part (a), `origin` which is a string representing the origin city, `destination` which is representing the destination city, and `weight` which is an integer representing the flight duration or flight distance depending on the graph sent as argument. This creates a direct connection between the origin and destination city, and adds the weight to the connection. If any of the origin or destination cities are not present in the network, it should print a message stating that the particular city not accessed by the flight network. If the connection already exists, it just updates the weight of the connection.

(f) `def add_airport(graph, city, destination, weight)` takes four parameters: `graph` which is the graph created in part (a), `city` which is a string representing the new destination being added to the existing graph, `destination` which is a string representing the city with which the new airport will have at least one connection, and `weight` which is an integer representing the flight duration or flight distance for that city with the destination. If the city is already in graph, just print a message stating that the airport already exists. Otherwise, it creates a new entry in the graph for the new airport.

(g) `def get_secondary_flights(graph,city)` which takes two parameters: `graph` which is the graph created in part (a), and `city` for which the secondary connections need to be checked. The function will first look for the immediate connections to that city, and from those immediate connections, it will get a list of all the connecting flights. The function returns a unique list of cities further connected to the immediate connection. This may or may not include the given `city`. Please note, we are only looking at the outbound connections. If the city does not exist, it returns None.

(h) `def counting_common_airports(graph,cityA,cityB)` which takes three parameters: `graph` which is the graph created in part (a), `cityA`which is a string representing one of the two cities, and `cityB` which is also representing the other city. The function counts the number of common airports in the two cities' connections and returns the count. If none common, it returns 0

(i) `def remove_flight(graph,origin,destination)` which takes three parameters: `graph` which is the graph created in part (a), `origin` which is a string representing the origin city, and `destination` which is a string representing the destination city. The function simply removes the connection between the two cities. If any of the origin or destination cities are not present in the network, it should print a message stating that the particular city not accessed by the flight network. Don't forget that these fights exist in both the graphs.

(j) `def remove_airport(graph,city)` which takes two parameters: `graph` which is the graph created in part (a), and `city` which is a string representing the

city being removed from the flight network. The function removes the airport city and all its connections from the graph if it exists. If the given city is not present in the network, it should print a message stating that the particular city not accessed by the flight network. Don't forget that these fights exist in both the graphs.

(k) `def find_all_paths(graph,origin,destination)` takes three parameters: `graph` which is the graph created in part (a), `origin` which is a string representing the origin city, and `destination` which is a string representing the destination city. The function then finds all the flight routes connecting the origin city to the destination, as nested list. If the origin and destination city are the same, it doesn't generate any path, i.e. returns an empty list. If the origin or destination city does not exist in the network, it returns None.

(l) `def find_number_of_layovers(graph,origin,destination)` which takes three parameters: graph which is the graph created in part (a), origin which is a string representing the origin city, and destination which is a string representing the destination city. The function then finds all the possible layovers when travelling from origin city to the destination city, in the form of a list sorted in ascending order. Non-stop flights will have 0 layover. If the origin and destination city are the same, as there is no path, therefore, it returns an empty list. If the origin or destination city does not exist in the network, it returns None

2.3.3 Input Format

The data set is defined in a CSV file called `flight_network.csv`, which contains four columns:

1. Origin City
2. Destination City
3. Flight Duration (in minutes)
4. Flight Distance (in miles)

The given dataset will produce two separate graphs:

- One graph will have vertices containing the city name, and edges will have flight duration between the two connected cities as their weight.
- The second graph will also have vertices as cities, and edges will have flight distance as their weights.

2.3.4 Output

In this question, the `main` function does not return any thing or write to an output file. The output of each function is dependent upon its return type.

Credits: Maria Samad