NHH Norwegian School of Economics

BAN401

Applied Programming and Data Analysis for Business

Final Group-based Project Report

Group 8

Kai Jing Zhou	S195385
Sean Zhi Yuan Tan	S195200
Jimeno Bianca Camille Mortel	S195206

Problem 1.1

Code

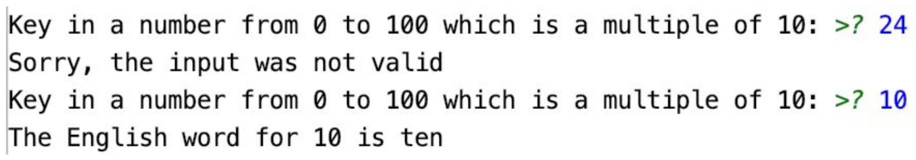
```
# Numbers and their corresponding words are stored in a dictionary
Numbers = {0:'zero', 10: 'ten', 20: 'twenty', 30: 'thirty', 40: 'forty',
     50: 'fifty', 60: 'sixty', 70: 'seventy', 80: 'eighty', 90: 'ninety', 100: 'hundred'}

# Prompting user to input a number
number_input = input('Key in a number from 0 to 100 which is a multiple of 10: ')

# Prompt the user for a number in the dictionary until a correct input is entered. Then, print the
  corresponding word of the number input.
while number_input not in [str(x) for x in Numbers.keys()]:
  print('Sorry, the input was not valid')
  number_input = input('Key in a number from 0 to 100 which is a multiple of 10: ')

print('The English word for ' + str(number_input) + ' is ' + Numbers[int(number_input)])
```

Explanation

We first created a dictionary named ‘Numbers’ with the keys being the numbers and the values being their associated words in English. The user is then prompted to input any number which is a multiple of 10 from 0 to 100.

The while loop is used to tell the user that his input is invalid and will prompt him to input another value. The loop only breaks when the user inputs a value that is one of the keys in the ‘Numbers’ dictionary. Once broken, the statement with the corresponding word of the number input is printed. This is done by accessing the value corresponding to the key which matches the user’s input in the ‘Numbers’ dictionary.

Result



```
Key in a number from 0 to 100 which is a multiple of 10: >? 24
Sorry, the input was not valid
Key in a number from 0 to 100 which is a multiple of 10: >? 10
The English word for 10 is ten
```

Problem 1.2

Code

```python
# Prompt the user for a sequence of words
seq = input('Enter any sequence of words: ')

# Splits the sequence so that words are elements in a list and set() removes repetition
new_seq = set(seq.split())

# Sorts the new sequence and joins the elements to form a string
new_seq = ' '.join(sorted(new_seq))

# Prints out the sorted sequence
print(new_seq)
```

Explanation

The user is prompted to enter any sequence of words. This input is stored in ‘seq’ and may contain uppercase/lowercase letters, repeated words and multiple spaces in between words.

‘new_seq’ is created which uses the split() and set() function. The split() function converts ‘seq’ from a string to a list containing the words as elements and also removes any spaces. The set() function then removes any repetition of elements.

Using the sorted() function, the elements are arranged in ascending order, arranging words that start with an uppercase letter first. The join() function then joins the elements to form a string, using ‘ ’ as a separator. Finally, ‘new_seq’ is printed to show the sorted sequence.

Result

```
Enter any sequence of words: >? Write Python Code  a sequence word each word can be Separated
Code Python Separated Write a be can each sequence word
```

Problem 2

Code

```
### Function to get the new chain
def get_the_chain(numbers):

  ## Creating a new list sorted in ascending order with no duplicates
  # Remove duplicates
  global i
  new_numbers = []  # Empty list where non-repeated numbers are added
  for i in range(0, len(numbers)):
    if numbers[i] not in new_numbers:  # If number from original list not in 'new_numbers', add it in
      new_numbers.append(numbers[i])

  # Sort 'new_numbers' in ascending order
  for j in range(1, len(new_numbers)):       # Iterate 'new_numbers' starting from the 2nd index
    # For each iteration, iterate 'new_numbers' starting from the 1st index
    for k in range(0, len(new_numbers)):
      if new_numbers[j] > new_numbers[k]:
        continue
      else:
        new_numbers[j], new_numbers[k] = new_numbers[k], new_numbers[j]

  ## Finding consecutive chains
  all_chains = [] #List containing all consecutive chains
  num_chain = []
  index1 = 1

  for n in new_numbers:
    # Check if current chain is empty or if current number is a consecutive number in the current chain
    if (len(num_chain) == 0) or (num_chain[-1] + 1 == n):
      num_chain += [n] # Add the number to current chain

      # Updates current chain in 'all_chains'
      if index1 == len(new_numbers):
        all_chains += [num_chain]
      index1 += 1

    # If current number is not consecutive, a new 'num_chain' is created beginning with current number
    else:
      all_chains += [num_chain]
      num_chain = [n]

      # Updates current chain in 'all_chains'
      if index1 == len(new_numbers):
        all_chains += [num_chain]
      index1 += 1
  ## Finding the longest consecutive chain that starts with the highest number
  # Find the lengths of all consecutive chains
  all_length = []
  for i in all_chains:
    all_length += [len(i)]

  # Find the index of the longest chain
  max = all_length[0]
  max_index = 0

  for l in range(0, len(all_length)):
    # If length in this index is greater than current highest length, update max and max_index
    if all_length[l] >= max:
      max = all_length[l]
      max_index = l # Index of the max length

  ## Print required lists
  print('Initial number list: ', numbers)
  print('Resulting number list: ', all_chains[max_index])


### List where function will be applied to
num_list = [0, 7, 4, 8, 1, 3, 8, 10, 11, 2, 5, 12, 9]

get_the_chain(num_list)
```

Explanation

To solve the problem, we create the **get_the_chain()** function with the list of numbers as the input. This function consists of 3 parts:

Creating a new list sorted in ascending order with no duplicates

Firstly, we need to create a new list which removes any duplicates and sorts the numbers in ascending order. We need to sort it in ascending order to make it easier to find the consecutive chains later on.

To remove the duplicates, we create an empty list ‘new_numbers’. We go through each number in the original list to check if it is already in ‘new_numbers’. If not, it is added inside. We now sort ‘new_numbers’ in ascending order. This is done by first comparing the second number with the first number. If the value of the second number is greater than the first number, the positions are ascending and therefore, the positions remain unchanged. If not, they exchange positions. This is done for the third and second number, fourth and third number and so on until all the numbers have been compared.

Finding consecutive chains

We create two empty lists, ‘all_chains’ and ‘num_chain’. New chains will be added in ‘all_chains’ while ‘num_chain’ will change with every iteration.

We iterate every number in ‘new_numbers’ until we obtain all consecutive chains. For each iteration, there are 2 scenarios:

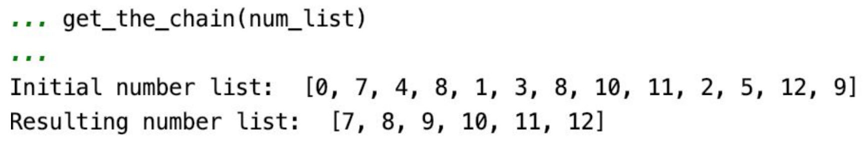
1. Consecutive numbers:
   If ‘num_chain’ is currently empty or if the current number in the iteration follows the last number in ‘num_chain’ (i.e. they are consecutive numbers), that number is added to ‘num_chain’.
2. Non-consecutive numbers:
   If ‘num_chain’ is not empty and the current number in the iteration does not follow the last number in ‘num_chain (i.e. they are not consecutive numbers), ‘num_chain’ is added to ‘all_chains’. This current chain ends and ‘num_chain’ is restarted with the current number as the first number.

Finding the longest consecutive chain that starts with the highest number
We find the lengths of all the consecutive chains and store it in ‘all_length’. To find the index of the longest chain, we begin by creating ‘max_index’ with a value of 0 and ‘max’ containing the value of the first element in ‘all_length’.

We then go through all the lengths in ‘all_length’ and compare it with the current ‘max’. If that length is greater or equal than ‘max’, we update ‘max_index’ with the index of the current iteration. When the for-loop ends, we will have the index of the longest consecutive chain whose first number is the highest.

Finally, we print the original list and then the final list, by extracting it from ‘all_chains’ using ‘max_index’.

Result



```
... get_the_chain(num_list)
...
Initial number list:  [0, 7, 4, 8, 1, 3, 8, 10, 11, 2, 5, 12, 9]
Resulting number list:  [7, 8, 9, 10, 11, 12]
```

Problem 3

Code

```
### Storing all the student data in different lists
first_name = ['Mike', 'Nancy', 'Steve', 'Mike', 'Jeffrey', 'Ivan', 'Sterling']
last_name = ['Wheeler', 'Wheeler', 'Harrington', 'Wazowski', 'Lebowski', 'Belik', 'Archer']
ids = ['19710', '19670', '19660', '18119', '69420', '12345', '11007']
gpa = [3.5, 3.6, 2.4, 2.9, 4.2, 1.8, 2.7]
major = ['FIE', 'ENE', 'STR', 'BAN', 'BLZ', 'BAN', 'MBM']
groups = ['it.gruppen', ['K7 Bulletin', 'NHHS Opptur', 'NHHS Energi'], 'N/A', 'N/A', ['NHHI Bowling',
'NHHI Vinum'], ['it.gruppen', 'NHHS Consulting'], 'NHHI Lacrosse']

### Function to retrieve info
def retrieval(i):
  print('Retrieving data for student {} {} (student ID {}).'.format(first_name[i], last_name[i], str(ids[i])))
  print('- GPA: ' + str(gpa[i]))
  print('- Major: ' + major[i])
  print('- NHHS group membership: ')
  if isinstance(groups[i], list): # Checks if a student has multiple groups
     for group in groups[i]:
        print('  - ' + group)
  else:
     print('  - ' + groups[i])


### Function used when there is several matches
def several_match(prompt, indexes_one):
  if prompt == 'all':
     for index in indexes_one:
        print('-------------')
        retrieval(index)
  elif (int(prompt) <= len(indexes_one) and int(prompt) >= 1):
     print('-------------')
     retrieval(int(prompt) - 1)
  else:
     print('Incorrect input. Please try again.')
     print('-------------')
     prompt = input('Enter the number of the search result for which you want to retrieve the info or
enter all to print info for all matching results: \n')
     several_match(prompt, indexes_one)


### Function to inquire for new query
def new_query(execution):
  execution = execution.lower() # Changes input to lowercase
  if execution == 'n':
     print('Exiting the program...')
  elif execution == 'y':
        query()
    else:
        print('-------------')
        print('Incorrect input. Please try again.')
        print('-------------')
        execution = input('Would you like to make a new search? (y/n)\n')
        new_query(execution)

### Function which starts the query
def query():
    search = input('Who are you looking for?\n')
    search = search.title() # Changes input so the first letter of each word is capitalised
    search = search.split() # Splits the input so that words are elements in a list

    indexes_one = [] # Empty list where index of any match will be added if user enters 1 word
    indexes_two = [] # Empty list where index of any match will be added if user enters 2 words

    ## Matching
    if len(search) == 1: # Matching when user enters 1 word
        for i in range(len(first_name)): # Check if input matches any of the elements in the 'first_name',
'last_name' or 'ids' lists
            if search[0] == first_name[i]:
                indexes_one.append(i) # If there is a match, index will be added to 'indexes_one'
            if search[0] == last_name[i]:
                indexes_one.append(i)
            if search[0] == ids[i]:
                indexes_one.append(i)

    if len(search) == 2: #Matching when user enters 2 words
        for i in range(len(first_name)): # Check if 1st word in input matches any of the elements in the
'first_name', 'last_name' or 'ids' lists
            if search[0] == first_name[i]:
                if search[1] == last_name[i]: # Check if 2nd word in input also matches
                    indexes_two.append(i)  #If there is a match, index will be added to 'indexes_two'
            elif search[0] == last_name[i]:
                if search[1] == first_name[i]:
                    indexes_two.append(i)

    ## Output
    if len(indexes_one) == 1 or len(indexes_two) == 1: # Output for one match
        print('-------------')
        print('One match found.')
        print('-------------')
        if len(indexes_one) == 1:
            retrieval(indexes_one[0])
        if len(indexes_two) == 1:
            retrieval(indexes_two[0])
```

```
  elif len(indexes_one) == 0 and len(indexes_two) == 0: # Output for no match
    print('No matches found.')

  elif (len(indexes_one) > 1 and len(indexes_two) == 0): # Output for several matches
    print('Several results matched your query:')
    for n,i in zip(range(len(indexes_one)), indexes_one):
      print('{}. {} {} (ID {})'.format(str(n+1), first_name[i], last_name[i], str(ids[i]))) # Prints out a
numbered list containing first name, last name and student IDs of all matches

    prompt = input('Enter the number of the search result for which you want to retrieve the info or
enter all to print info for all matching results: \n')
    several_match(prompt,indexes_one)

  ## Request for new query
  print('-------------')
  execution = input('Would you like to make a new search? (y/n)\n')
  new_query(execution)

query()
```

Explanation

We begin by storing all the student data in lists. Each type of information will have its own list. For first name, last name and student ID, they are stored as strings to make it easier to match with user input, which is already stored as a string by default.

We create a **query()** function which prompts users to input their search query. We then split the input so that their input becomes a list containing the words as elements. For example, if they input ‘Nancy’, the resulting list would be [‘Nancy’]. If they input ‘Ivan Belik’, the resulting list would be [‘Ivan’, ‘Belik’].

The query() function consists of 3 main parts - matching, output and request for new query.

Matching

To check for matches between the user input and Table 2, the general idea would be to check whether the length of the input list is 1 or 2.

If the length is 1, we check whether the input matches any of the first names, last names or student IDs. If there is a match, the index of this match will be added to ‘indexes_one’.

If the length is 2, the input is only valid if both first and last name are found in Table 2. We first check whether the input matches any of the first name or last name. If there is a match, we then check if it also matches the last name. For example, if the input is [‘Ivan’, ‘Belik’], we can see that it matches index 5 of ‘first_name’ and also index 5 of last name. If that is the case, the index of this match will be added to ‘indexes_two’.

Output
Once we have checked for matches, we need to produce the corresponding output to show the user. Since an input may match multiple students, there are 3 possible outputs:

1. 1 match:
   If the length of ‘indexes_one’ or ‘indexes_two’ is 1, this means that there is only 1 index that matches the input and hence, there is only one match. We then run the retrieval() function with the only index in ‘indexes_one’ or ‘indexes_two’ as the input.

   The **retrieval()** function uses the aforementioned index to extract elements from all the lists we created at the start. For the list ‘groups’, a student may have multiple groups so a condition is created where a for-loop is used to ensure that all groups are printed, should a student have multiple groups.

2. No match:
   If the length of ‘indexes_one’ or ‘indexes_two’ is 1, there is no match. We then print ‘No matches found.’

3. Several matches:
   If the length of ‘indexes_one’ is more than 1, there are several matches. We then print the first name, last name and student IDs of the matches and numbering each result, starting from 1. The user is prompted to ‘Enter the number of the search result for which you want to retrieve the info or enter all to print info for all matching results’. We then run the several_match() function with the user’s input and the indexes from ‘indexes_one’ as the inputs.

   The **several_match()** function has 3 different scenarios:
   1. If user inputs ‘all’, the ‘retrieval()’ function is run with all the indexes in ‘indexes_one’ as the input.
   2. If user inputs a number that is within 1 and the length of ‘indexes_one’, the ‘retrieval()’ function is run with the specified index as the input. In this case, if the user inputs 2, the index would be (2-1) since the numbering begins from 1 instead of 0.
   3. If user inputs a number that is less than 1 or greater than the length of ‘indexes_one’, their input is invali. They will then be prompted to enter another number and the several_match() function is run once again.

Request for New Query
Once an output has been created, the user will be asked if they would like to make a new search. The new_query() function is then run.

The **new_query()** function changes the user input to lowercase so regardless of whether they put ‘Y’ or ‘y’, the result will be the same. The function has 3 different scenarios:

1. If user inputs ‘n’, the whole query() function will be terminated.
2. If user inputs ‘y’, the query() function is run again.
3. If the user input is neither ‘n’ or ‘y’, their input is invalid and they will be asked whether they would like to make a new search and the new_query() function will be ran again.

Result

(3)

```
... query()
...
Who are you looking for?
>? Wheeler
Several results matched your query:
1. Mike Wheeler (ID 19710)
2. Nancy Wheeler (ID 19670)
Enter the number of the search result for which you want to retrieve the info or enter all to print info for all matching results:
>? 2
-------------
Retrieving data for student Nancy Wheeler (student ID 19670).
- GPA: 3.6
- Major: ENE
- NHHS group membership:
  - K7 Bulletin
  - NHHS Opptur
  - NHHS Energi
-------------
Would you like to make a new search? (y/n)
>? y
Who are you looking for?
>? Ivan Belik
-------------
One match found.
-------------
Retrieving data for student Ivan Belik (student ID 12345).
- GPA: 1.8
- Major: BAN
- NHHS group membership:
  - it.gruppen
  - NHHS Consulting
-------------
Would you like to make a new search? (y/n)
>? y
Who are you looking for?
>? Jeffrey
-------------
One match found.
-------------

Retrieving data for student Jeffrey Lebowski (student ID 69420).
- GPA: 4.2
- Major: BLZ
- NHHS group membership:
  - NHHI Bowling
  - NHHI Vinum
-------------
Would you like to make a new search? (y/n)
>? That rug really tied the room together!
-------------
Incorrect input. Please try again.
-------------
Would you like to make a new search? (y/n)
>? n
Exiting the program...

>>>
```

(4)

```
>>> query()
Who are you looking for?
>? Mike
Several results matched your query:
1. Mike Wheeler (ID 19710)
2. Mike Wazowski (ID 18119)
Enter the number of the search result for which you want to retrieve the info or enter all to print info for all matching results:
>? all
-------------
Retrieving data for student Mike Wheeler (student ID 19710).
- GPA: 3.5
- Major: FIE
- NHHS group membership:
  - it.gruppen
-------------
Retrieving data for student Mike Wazowski (student ID 18119).
- GPA: 2.9
- Major: BAN
- NHHS group membership:
  - N/A
-------------
Would you like to make a new search? (y/n)
>? y
Who are you looking for?
>? 11007
-------------
One match found.
-------------
Retrieving data for student Sterling Archer (student ID 11007).
- GPA: 2.7
- Major: MBM
- NHHS group membership:
  - NHHI Lacrosse
-------------
Would you like to make a new search? (y/n)
>? y
Who are you looking for?
>? Copernicus
No matches found.

>>>
```

Problem 4

Code

```r
# Set working directory if the current working directory is not correctly set
if  (getwd()  !=  "C:/Users/Kai  Jing/Desktop/NUS/Business  (Accountancy)/NUS
BAC/Sem 3.1/BAN401 - Applied Programming and Data Analysis For Business/Final
Project"){
    setwd("C:/Users/Kai  Jing/Desktop/NUS/Business  (Accountancy)/NUS  BAC/Sem
3.1/BAN401  -  Applied  Programming  and  Data  Analysis  For  Business/Final
Project")
}

# Clear workspace
rm(list = ls())

### Problem 4
# Create a list of all the available denominations
Card.Points <- c(1, 2, 5, 10, 20, 50, 100)

# Set the base case for the number of combinations where no denominations of
points are used
Base.Case <- c(c(1), rep(0, 200))

# Create a matrix detailing the number of combinations for all the points
leading up to 200
# Then, fill in the matrix with number of combinations for that total points
(in the column)
# by deciding if the corresponding denomination in the Points List is used or
not
Matrix.Sol <- matrix(nrow = 7, ncol = 201, byrow = T)
Matrix.Sol <- rbind(Base.Case, Matrix.Sol)
rownames(Matrix.Sol) <- c('{}',
                          '{1}',
                          '{1,2}',
                          '{1,2,5}',
                          '{1,2,5,10}',
                          '{1,2,5,10,20}',
                          '{1,2,5,10,20,50}',
                          '{1,2,5,10,20,50,100}')
colnames(Matrix.Sol) <- 0:200

for (i in 2:nrow(Matrix.Sol)){
  for (j in 1:ncol(Matrix.Sol)){
    if (j-1 < Card.Points[i-1]){
      Matrix.Sol[i,j] <- Matrix.Sol[i-1,j]
    } else{
```

```
                                Matrix.Sol[i,j]     <-     Matrix.Sol[i-1,j]     +
Matrix.Sol[i,(j-Card.Points[i-1])]
     }
   }
}

cat('There are', as.character(Matrix.Sol[nrow(Matrix.Sol), ncol(Matrix.Sol)]),
'possible scenarios to buy one "2 megapoints"-card.')
```

Explanation

We begin by setting the correct current working directory (to the folder containing the *.R* code) if the current working directory is not set correctly, and clearing the workspace of all objects and variables.

Intuition

To approach this problem, we first look at all the set of cards by considering whether each individual card is used or not. To do that, we consider the possible set-of-card(s) scenarios, starting from the set with no cards {}, the set with only the “1 point”-card {1}, the set with “1 point” and “2 point” cards {1,2}, and so on, all the way up to the set with all cards up to the “1 megapoint”-card {1,2,5,10,20,50,100}.

To find the total number of combinations that exists for the full set of cards {1,2,5,10,20,50,100} for “2 megapoints”, there are two scenarios to consider in the first step, one where the “1 megapoint”-card is used {1,2,5,10,20,50,100} and one where the “1 megapoint”-card is not used {1,2,5,10,20,50}. The total number of combinations that make up “2 megapoints” from the full set of cards is simply the sum of the combinations given by these two scenarios.

For the second step, to determine the number of combinations for “2 megapoints” when the “1 megapoint”-card is used {1,2,5,10,20,50,100}, we have to now determine the combinations for “1 megapoint” (2 megapoints – 1 megapoint = 1 megapoint, since the “1 megapoint”-card is used) using the same full set of cards {1,2,5,10,20,50,100}. This can be further broken down to the similar two scenarios as before, where the “1 megapoint”-card is used {1,2,5,10,20,50,100} or not {1,2,5,10,20,50}. The total number of combinations that make up “1 megapoint” from the full set of cards is simply the sum of the combinations given by these two scenarios.

For the third step, to determine the number of combinations for “2 megapoints” when the “1 megapoint”-card is not used {1,2,5,10,20,50}, we can see that this also comprises two scenarios, one where the “50 points”-card is used {1,2,5,10,20,50} and one where it is not used {1,2,5,10,20}. The total number of combinations that make up “2 megapoints” from the set of cards {1,2,5,10,20,50} is simply the sum of the combinations given by these two scenarios.

Extending our above intuition all the way to the base case, we obtain the following general formula:

Table[row, column] = Table[row - 1, column] + Table[row, (column – Card.Points[row - 1])]

We also note that for each scenario where the value of the largest card in the set of cards exceeds the total points we are trying to achieve, the largest card cannot be used. Hence, the solution is simply the same as the combinations with the set of cards without that largest card. (E.g. combinations for 5 points using {1,2,5,10} is the same as combinations for 5 points using {1,2,5})

Given that, we have the following alternative general formula when this scenario occurs:

Table[row, column] = Table[row - 1, column]

<u>**Base Case**</u>

To obtain the base case of our solution, we determine that the set with no cards {}, gives us 1 combination for the scenario where we need to make 0 points, and no solution for all scenarios from 1 to 200 points.

<u>**Dynamic Programming**</u>

Now that we understand the problem, its parameters, and the intuition behind the solution, we can move on to the programming solution. To solve the problem, we need to create the following data structures:

1. Vector of all the card point denominations

2. Vector of solutions for the base case with the empty set of cards {}

3. Matrix to contain all the solutions for all the sets of cards other than the empty set, {1}, {1,2}, …, {1,2,5,10,20,50,100}, for all the points from 0 to 200

After creating the above data structures, we first create our full matrix, *Matrix.Sol*, by attaching our base case vector to our matrix for the remaining solution using *rbind()*. Then, we rename the matrix columns to reflect accurately the points we are trying to find for each column (since R uses 1-based indexing). We also rename the matrix row names to reflect the current set of cards for the scenarios in that row.

Next, we use a double *for*-loop to iterate (from top to bottom, then left to right) through the rest of the empty cells in the matrix and fill it up with the combinations for each scenario. An *if-else* conditional is used to check if the largest card in the current set of cards, given by *Card.Points(i – 1)*, exceeds the current points according to the column index (*j – 1*, since R uses 1-based indexing). If it does, as noted in the base case, the following formula applies:

Table[row, column] = Table[row - 1, column]

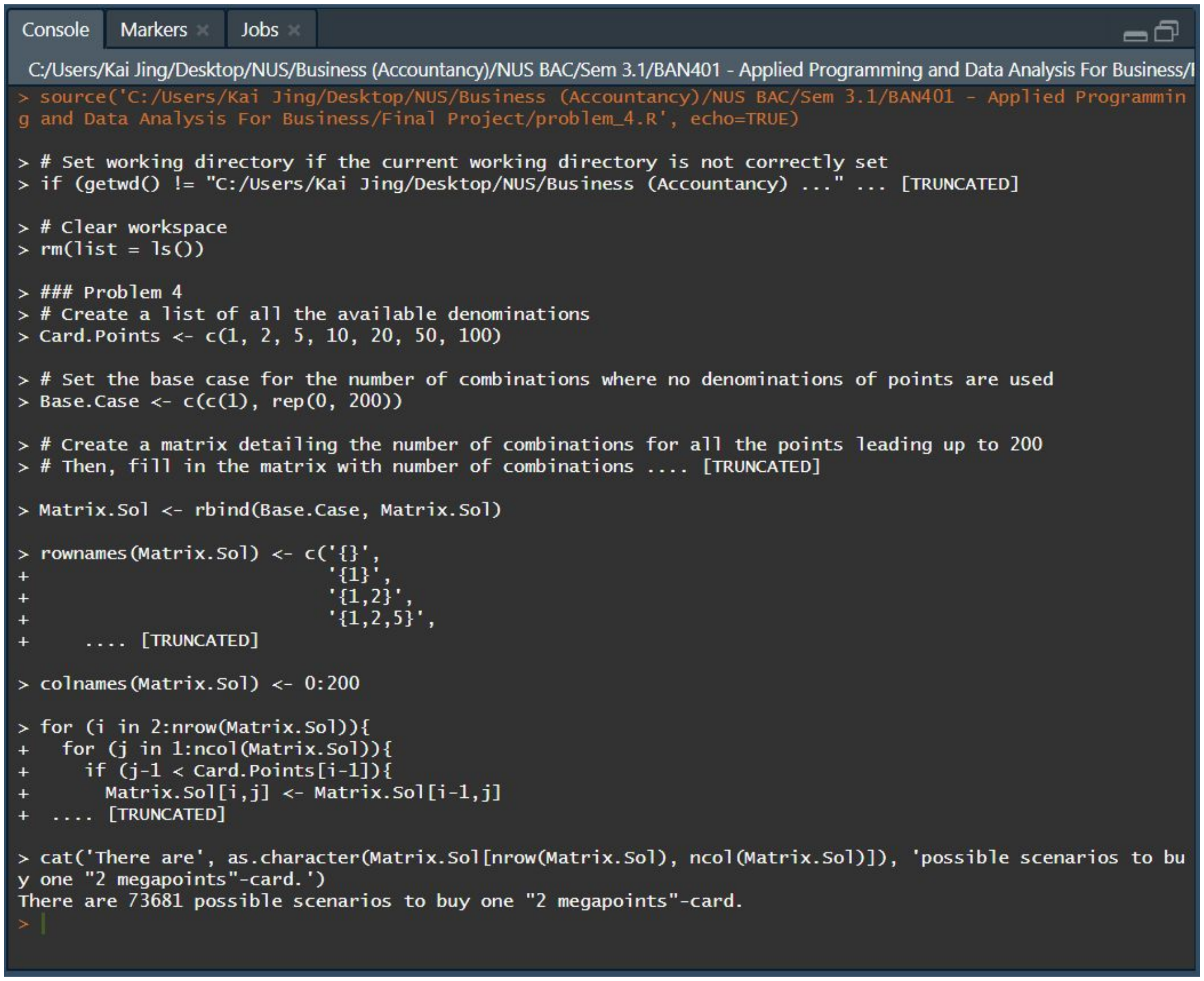
Otherwise, we apply the following formula:

Table[row, column] = Table[row - 1, column] + Table[row, (column – Card.Points[row - 1])]

**Card.Points[row - 1]* is used since our matrix starts with the empty set of cards {}

Finally, we extract the total number of combinations for 200 points with the full set of cards in the bottom right cell of the matrix using *Matrix.Sol[nrow(Matrix.Sol), ncol(Matrix.Sol)])* and use *cat()* to display the solution.

Result


Console
Markers
Jobs
C:/Users/Kai Jing/Desktop/NUS/Business (Accountancy)/NUS BAC/Sem 3.1/BAN401 - Applied Programming and Data Analysis For Business/

```
> source('C:/Users/Kai Jing/Desktop/NUS/Business (Accountancy)/NUS BAC/Sem 3.1/BAN401 - Applied Programmin
g and Data Analysis For Business/Final Project/problem_4.R', echo=TRUE)

> # Set working directory if the current working directory is not correctly set
> if (getwd() != "C:/Users/Kai Jing/Desktop/NUS/Business (Accountancy) ..." ... [TRUNCATED]

> # Clear workspace
> rm(list = ls())

> ### Problem 4
> # Create a list of all the available denominations
> Card.Points <- c(1, 2, 5, 10, 20, 50, 100)

> # Set the base case for the number of combinations where no denominations of points are used
> Base.Case <- c(c(1), rep(0, 200))

> # Create a matrix detailing the number of combinations for all the points leading up to 200
> # Then, fill in the matrix with number of combinations .... [TRUNCATED]

> Matrix.Sol <- rbind(Base.Case, Matrix.Sol)

> rownames(Matrix.Sol) <- c('{}',
+                           '{1}',
+                           '{1,2}',
+                           '{1,2,5}',
+     .... [TRUNCATED]

> colnames(Matrix.Sol) <- 0:200

> for (i in 2:nrow(Matrix.Sol)){
+   for (j in 1:ncol(Matrix.Sol)){
+     if (j-1 < Card.Points[i-1]){
+       Matrix.Sol[i,j] <- Matrix.Sol[i-1,j]
+   .... [TRUNCATED]

> cat('There are', as.character(Matrix.Sol[nrow(Matrix.Sol), ncol(Matrix.Sol)]), 'possible scenarios to bu
y one "2 megapoints"-card.')
There are 73681 possible scenarios to buy one "2 megapoints"-card.
>
```

Problem 5

Code

```
# Set working directory if the current working directory is not correctly set
if  (getwd()  !=  "C:/Users/Kai  Jing/Desktop/NUS/Business  (Accountancy)/NUS
BAC/Sem 3.1/BAN401 - Applied Programming and Data Analysis For Business/Final
Project"){
    setwd("C:/Users/Kai  Jing/Desktop/NUS/Business  (Accountancy)/NUS  BAC/Sem
3.1/BAN401  -  Applied  Programming  and  Data  Analysis  For  Business/Final
Project")
}

# Clear workspace
rm(list = ls())

### Problem 5
# Read csv files containing the data for Table 3 and Table 4, using ';' as a
separator
Table.3 <- read.csv('ban401_fuzzy-matching-table3.csv', sep = ';')
Table.4 <- read.csv('ban401_fuzzy-matching-table4.csv', sep = ';')

# Create column in Table 3 to store Total Granted Patents data
Table.5 <- Table.3
Table.5$Total.number.of.patents <- NA

#  Check  each  company  name  in  Table  3  against  Table  4's  company  names  in
'Applicant" column using agrep()
#  Then,  subset  Table  4's  Status  column  by  the  index  result  and  obtain  the
total number of "Granted" patents,
# and assign it to corresponding company in Table 3
for (i in 1:nrow(Table.5)){
  index <- agrep(Table.5$Company.name[i],
                 Table.4$Applicant,
                 ignore.case = T)
      Table.5$Total.number.of.patents[i]   <-   sum(Table.4$Status[index]   ==
"Granted")
}

Table.6 <- Table.4[Table.4$Status=='Granted',]
rownames(Table.6) <- NULL

for (i in 1:nrow(Table.6)){
  x <- agrep(Table.6$Applicant[i],Table.5$Company.name,ignore.case = T)

  if (length(x) != 0){
    Table.5$Total.number.of.patents[x] <- Table.5$Total.number.of.patents[x] +
1
  }
}

# Sort Table 3 according to the number of patents each company has in
descending order
Table.5 <- Table.5[order(Table.5$Total.number.of.patents, decreasing = T),]
rownames(Table.5) <- NULL

head(Table.5, 5)
```

Explanation

We begin by setting the correct current working directory (to the folder containing the *.R* code) if the current working directory is not set correctly, and clearing the workspace of all objects and variables.

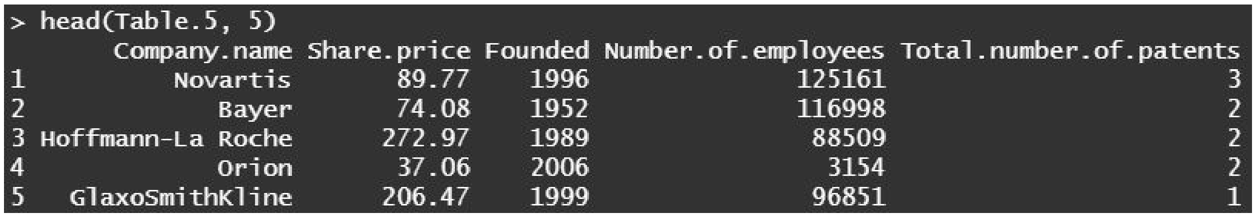
For problem 5, we first begin by reading the *.csv* files provided using the *read.csv()* function and assigning them into the variables *Table.3* and *Table.4* respectively. Per question requirements, we created another variable *Table.5* that is the same as *Table.3*, and added the column *Total.number.of.patents* with empty *NA* variables to be filled in later.

Using a *for*-loop and the *agrep()* function, we went through each of the company names in *Table.5* and matched it against the applicant names in *Table.4*. The *ignore.case* = *T* parameter is used to facilitate matching since all *Table.4$Applicant* values are in uppercase. Then, we searched for all the *"Granted"* status in our matches and the *sum()* of the resting logical vector match gives us the total number of patents granted for each company.

However, this match is incomplete as *agrep()* fails to capture the granted patent for *"GlaxoSmithKline"* (*"SmithKline"* in *Table.4$Applicants*) since *"GlaxoSmithKline"* failed to match against *"SmithKline"* pattern is not a subset of the item being matched against. Hence, we performed another match in the reverse direction from *Table.4$Applicant* to *Table.5$Company.name* using the *agrep()* function after filtering for the *"Granted"* status. If there are any matches, we would add 1 to the total patent count for that company in *Table.5*.

Finally, we arranged *Table.5* in decreasing order of total patents using the *order()* function and by setting *decreasing* = *T* as a parameter before printing out *Table.*5's top 5 rows as the required result.

Result



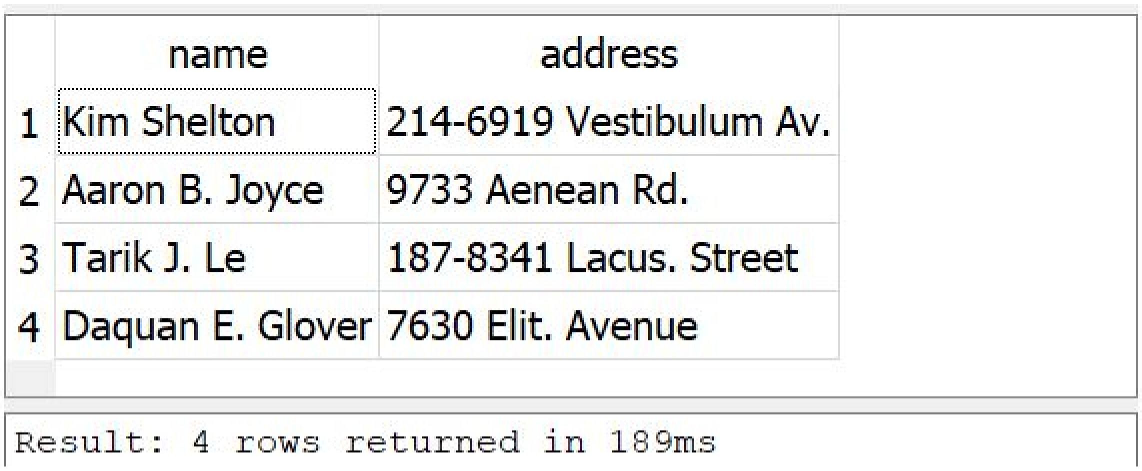
```
> head(Table.5, 5)
      Company.name Share.price Founded Number.of.employees Total.number.of.patents
1         Novartis       89.77    1996              125161                       3
2            Bayer       74.08    1952              116998                       2
3 Hoffmann-La Roche     272.97    1989               88509                       2
4            Orion       37.06    2006                3154                       2
5   GlaxoSmithKline     206.47    1999               96851                       1
```

Problem 6

Code

```
SELECT name,address FROM Customers_Info c
        INNER JOIN Orders o
                ON c.customer_id = o.customer_id
        INNER JOIN Items_from_order i
                ON o.order_id = i.order_id
        WHERE i.product_name = 'Lime';
```

Result



	name	address
1	Kim Shelton	214-6919 Vestibulum Av.
2	Aaron B. Joyce	9733 Aenean Rd.
3	Tarik J. Le	187-8341 Lacus. Street
4	Daquan E. Glover	7630 Elit. Avenue

```
Result: 4 rows returned in 189ms
```

Problem 7

The first challenge would be the different platforms used for database and difficulty with access to data, due to the different platforms and the different firm-specific plugins used for information retrieval.

Company X and Company Y may have different information retrieval systems, either due to the way their data is structured, or relevancy of the information their users would seek.

For example, if Company X were a venture capital firm, information retrieval queries may be optimised to find the information most relevant to current market prices, to the current market environment or to past transactions the venture capital firm had carried out that has the most relevance to the current transactions they are working on.

On the other hand, if the company being acquired, Company Y, were a technology firm such as Spotify, their information retrieval system, as well as their database, would be radically different. This is because Spotify collects and processes data such as song lyrics, written lyrics and track metadata, which is radically different compared to the statistics that venture capital analysts would search for, which are queries for transaction multiples, purchase premiums or other relevant market multiples.

This challenge grows bigger as the disparity between the business lines or industry of the acquirer, Company X, and the one being acquired, Company Y, grow bigger. This problem would probably be seen in companies attempting acquisitions and integration of databases with other companies in industry horizontals. An example would be Google, at that time an internet service software company, acquiring Linkedin, which was at its core, a networking website.

In addition, should the databases be integrated, the information retrieval system might also have to be merged. As such, employees who were once used to operating on the information retrieval system that was phased out might not be able to work as efficiently as before. This temporary drop in productivity would only be able to be solved with time and training, adding additional costs, tangible and intangible, to the database integration.

The second challenge would be the difficulties in aligning the formatting of the data. This includes issues with firms storing data as structured or unstructured data, as well as storing what should be the same data type as INT in 1 firm and CHAR in another firm.

In the case of unstructured vs structured data, we can first assume that Company X is an established company, with its data stored as structured data, establishing relational models for easier retrieval, access and manipulation.

However, Company Y may be a company that stores data in an unstructured format. For example, Company Y may be a company dealing mainly with reports, such as a small financial audit firm or a law firm . This would mean a majority of their database records would be documents of the word format, which would be difficult to integrate into Company X's structured data databases.

Furthermore, Company X may store certain records in the INT form, but Company Y might have stored the same records in the CHAR form. This might result in loss of data upon integrating both companies' databases, due to data type conversion.

The third challenge would be anomalies in the updating process, after the formatting has been aligned and databases merged. These anomalies take 3 forms, namely: Update Anomaly; Insert Anomaly and Delete Anomaly.

We assume that Company X and Company Y are now using the same integrated database.

Firstly, the Update Anomaly. The update anomaly occurs when there is an update to the database, but the corresponding columns or rows of the table are not updated properly, resulting in incorrect information. For example, the Merged Company (of Company X and Company Y) may choose to change the location of their regional headquarters to a location more convenient for both employees of X and Y to access, such as moving their headquarters from Bergen to Oslo. However, when updating the location of their headquarters, the respective departments who moved with the regional headquarters may still have their location in Bergen, which is incorrect, as their location, along with the location of regional headquarters, should be in Oslo. Although incorrect address information might not be very important for day-to-day tasks, it could also result in sensitive information being delivered or couriered to the wrong offices, which might result in information leakage or parties obtaining information that they should not have, either about the company or about circumstances surrounding the company.

Next is the Insert Anomaly. This occurs when adding a new, unique record that requires data that does not exist. An example is the addition of a sales order. The currently integrated database of Merged Company might require that a customer is recorded first, before a sale to the customer can be recorded.

However, if that customer is a new customer to Merged Company, Merged Company would not have any existing records in their database about that customer.

Likewise, Merged Company's integrated database may require that an invoice number be present to add a customer record, but with a new customer that has yet to purchase anything, no invoice number can be recorded, and the data on the new customer is prevented from being entered into the integrated database.

Finally, is the Delete Anomaly. This anomaly occurs when deleting data that contains attributes that should not be deleted. An example would be when a supplier of goods (Supplier A) to the Merged Company does not have any outstanding deliveries to be made to the Merged Company. Deletion of this supplier from a table of “outstanding deliveries” or “deliveries yet to be received” may on the surface appear to be sound. However, in reality, deleting this supplier from those tables may also delete all relevant information about the supplier in other related tables. This would result in loss of data on the Merged Company’s end. Merged Company’s tracking of that supplier, its past payments as well as deliveries received on time and goods supplied would all be lost. This loss of data would mean that Merged Company would not be able to track the performance or reliability of this supplier. In addition, if the Merged Company might want to retrieve the lost data for record-keeping purposes, they might have to request the data from supplier A, which might either not be reliable, or might not be recorded at all.