CS1010S — Programming Methodology
School of Computing
National University of Singapore

Solutions for Mock Midterm

26 February 2024 **Time allowed:** 1 hour 20 minutes

Matriculation No: | | | | | | | | | |

Instructions (please read carefully):

1. Write down your matriculation number on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is **an open-sheet quiz**. You are allowed to bring one A4 sheet of notes (written on both sides).
3. This paper comprises **FOUR (4) questions** and **TWENTY-ONE (21) pages**. The time allowed for solving this mock midterm is **1 hour 20 minutes**.
4. The maximum score of this mock midterm is **60 marks**. The weight of each question is given in square brackets beside the question number.
5. All questions must be answered correctly for the maximum score to be attained.
6. All questions must be answered in the space provided in the answer sheet; no extra sheets will be accepted as answers.
7. The back-sides of the sheets and the pages marked "scratch paper" in the question set may be used as scratch paper.
8. You are allowed to un-staple the sheets while you solve the questions. Please make sure you staple them back in the right order at the end of the mock midterm.
9. You are allowed to use pencils, ball-pens or fountain pens, as you like (no red color, please).
10. Use of calculators are not allowed in the test.

Disclaimer: This unofficial mock midterm was created by TA Zhu Ming, TA Brian, and TA Russell. It does not reflect the standards of an official CS1010S midterm.

GOOD LUCK!

Question	Marks	Remark
Q1		
Q2		
Q3		
Q4		
Total		

Question 1: Python Expressions [17 marks]

There are several parts to this problem. Answer each part **independently and separately**. In each part, one or more Python expressions are entered into the interpreter (Python shell). If the interpreter produces an error message, or enters an infinite loop, explain why. For diagrams, you are expected to demonstrate the techinques you have learnt in class.

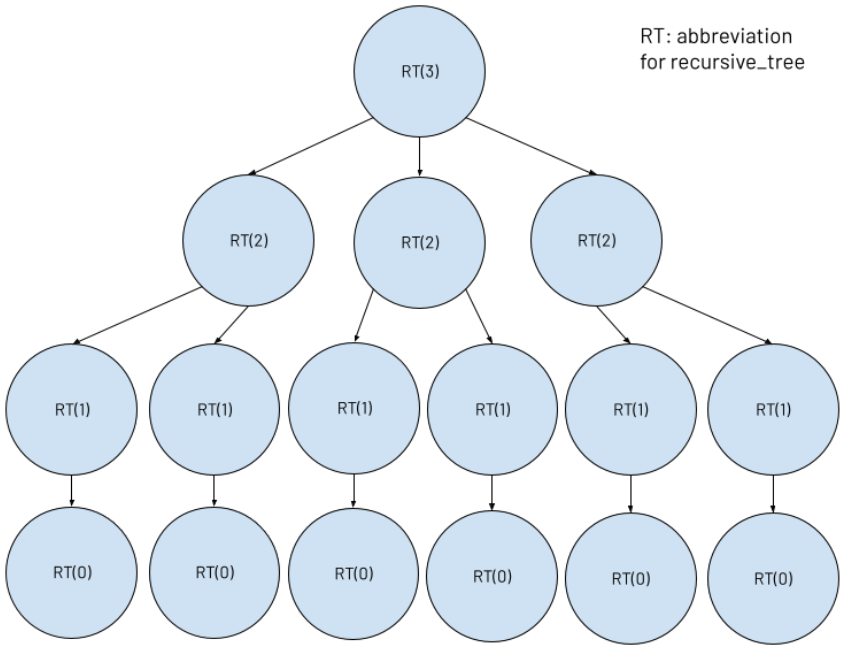
Consider the following definition of `recursive_tree` for parts **A** and **B**.

```
def recursive_tree(n):
    if n == 0:
        return 1
    else:
        for i in range(0, n, 1):
            recursive_tree(n-1)
```

A. Draw the recursive tree model for the function call `recursive_tree(3)`.

[3 marks]

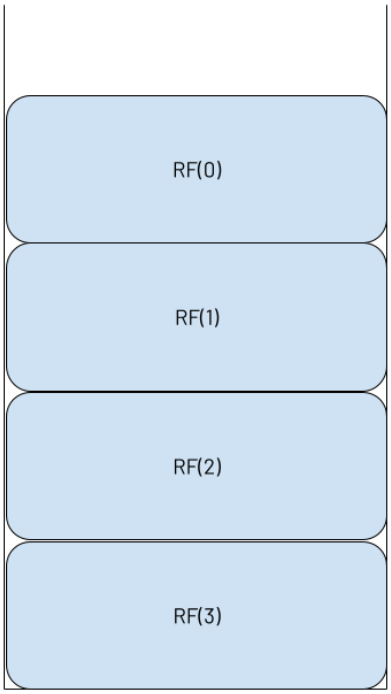
This question tests if the student understands the recursive tree model of a recursive function call.



B. Draw the call stack when the function call `recursive_tree(3)` **reaches the base case the first time**.

[3 marks]

This question tests if the student understands the call stack model of a recursive function call.



C. Draw the trace table for the following code and determine the output clearly:

```
x = "12345678901"
res = "0"
for i in range(2, 9, 1):
    if int(x[i]) % 3 == 0:
        res = x[i]
        print(res)
    elif i > 5:
        print(res)
        continue
    else:
        res = x[-1 * i]
        print(res)
print(i)
```

[3 marks]

Trace table:

i	res	print	x[i]	x[-1*i]
2	"3"	3	"3"	-
3	"9"	9	"4"	"9"
4	"8"	8	"5"	'8"
5	"6"	6	"6"	-
6	"6"	6	"7"	-
7	"6"	6	"8"	-
8	"9"	9	"9"	-

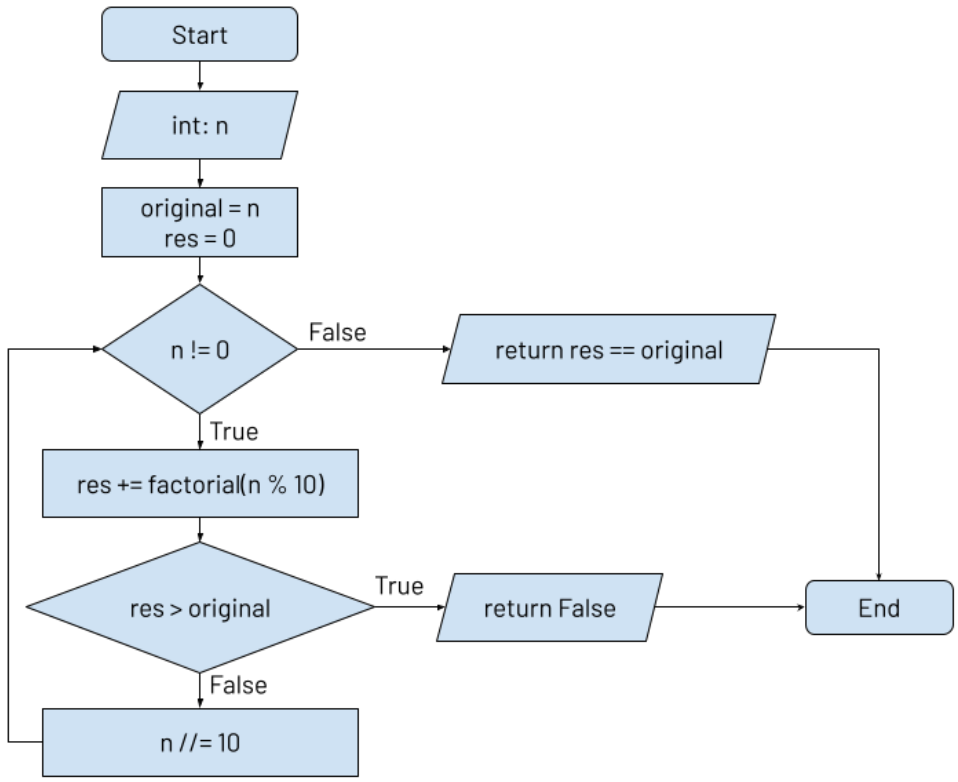
The output is:

```
3
9
8
6
6
6
9
8
```

This question tests if the student understands the trace table model of a simple code tracing question. The first two column `i` and `res` of trace table is mandatory because they trace the state of the variables. The remaining columns are optional and serve as helper columns.

D. Given the Polya flowchart below, convert it to a Python code. You may assume that the function `factorial(n)` has been provided to you. `factorial(n)` returns the factorial of n.

[2 marks]




```
def special_number(n):
    original = n
    res = 0

    while n != 0:
        res += factorial(n % 10)
        if res > original:
            return False
        n = n // 10
    return res == original
```

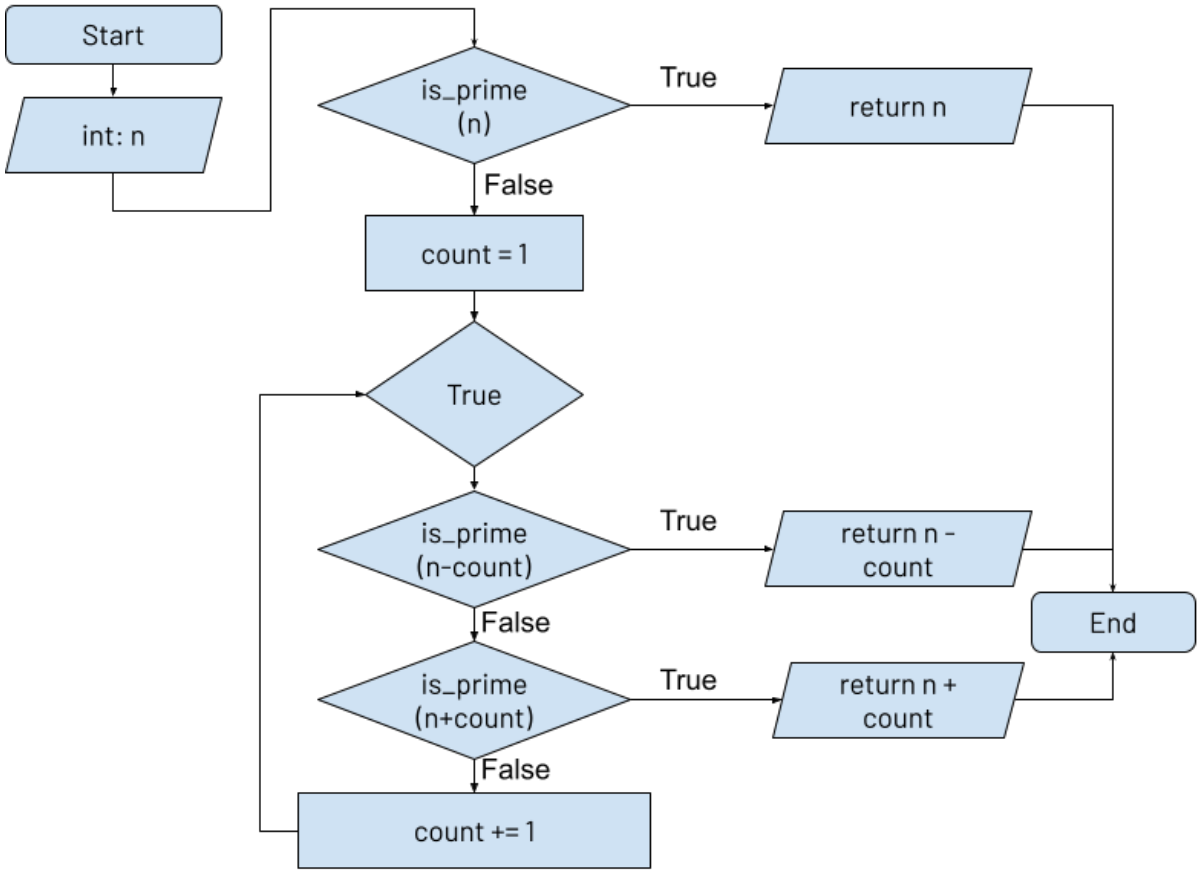
This question tests if the student understands the conversion to a Python code from a working Polya flowchart.
This code determines whether a given number equals the sum of the factorials of its individual digits. For example, $145 = 1! + 4! + 5! = 1 + 24 + 120 = 145$ *is a special number.*

E. Given the Python code below, construct a Polya flowchart of it. You may assume that the function `is_prime(n)` has been provided to you. `is_prime(n)` returns `True` if `n` is a prime number, and `False` otherwise.

```
def nearest_prime(n):
    if is_prime(n):
        return n
    count = 1
    while True:
        if is_prime(n - count):
            return n - count
        elif is_prime(n + count):
            return n + count
        count += 1
```

[3 marks]

This question tests if the student understands the conversion to a Polya flowchart from a working Python code.



F. Determine the output of the following Python code:

```
x = 10
new_x = -x
this_x = new_x + x
that_x = new_x - x

def why_so_many_x(takes_in_x):
    my_x = this_x
    your_x = that_x
    same_x = takes_in_x
    def new_x(x):
        if your_x is my_x:
            print("same x!")
            return same_x
        elif this_x is that_x:
            print("which x!")
            return x
        else:
            print("takes in x!")
            return takes_in_x
    if new_x(same_x) is takes_in_x:
        print("new_x is takes_in_x!")
        return x
    else:
        print("new_x is not takes_in_x!")
        return new_x(takes_in_x)

print(why_so_many_x(x))
```

[3 marks]

```
takes_in_x!
new_x is takes_in_x!
10
```

This question tests if the student understands the concept of variable scope.

Question 2: Datetimes [8 marks]

Warning: You should **NOT** use any string-related methods such as `.split`, `.find`, `.join`. You may assume that the function `substring(string, start, end, step)` has been provided to you and you should **NOT** use slice operator.

A. Implement a Python function `later_date` that takes in two dates given in the form of **YYYY-MM-DD** strings (also known as the ISO format). The function should return the first date if the first date is larger than the second date, the second date if the second date is larger than the first date, and either date if the two dates are equal. You should **NOT** use any built-in functions such as `max` and `min`.

Here's a sample execution:

```
>>> later_date("2024-02-08", "2023-01-12")
"2024-02-08"
>>> later_date("2023-01-12", "2023-01-12")
"2023-01-12"
>>> later_date("2024-02-26", "2025-10-01")
"2025-10-01"
```

Hint: You can just *directly compare* the strings. This is what makes the ISO format awesome!

[3 marks]

```
def later_date(date1, date2):
    if date1 > date2:
        return date1
    else:
        return date2
```

B. Zhu Ming realizes that the dates given are apparently not always in the **YYYY-MM-DD** format, but the **DD-MM-YYYY** format instead. Brian, not caring about the ongoing issue, still wants Zhu Ming to find the later date between two dates.

Implement a function `to_iso` that takes in a date in the form of EITHER **YYYY-MM-DD** or **DD-MM-YYYY**, and returns the same date in the **YYYY-MM-DD** format.

You should **NOT** use any string-related methods such as `.split`, `.find`, `.join`. You may assume that the function `substring(string, start, end, step)` has been provided to you.

Here's a sample execution:

```
>>> to_iso("2024-10-18")
"2024-10-18"
>>> to_iso("01-02-2028")
"2028-02-01"
```

[3 marks]

```
def to_iso(date):
    if date[4] == "-":
        return date
    else:
        day = substring(date, 0, 2, 1)
        month = substring(date, 3, 5, 1)
        year = substring(date, 6, 10, 1)
        return year + "-" + month + "-" + day
```

C. Having implemented the `to_iso` function, Zhu Ming feels safer now. Now, there is one last thing to do.

You are to implement a function `better_later_date` that takes in two dates, each in either of the **YYYY-MM-DD** or the **DD-MM-YYYY** format, and returns the later date among the two in the form of **YYYY-MM-DD**.

Being the generous of a person, Brian has given you the template of the code. In return, you simply need to fill in the blank with just **ONE** line of code, i.e. `<T3>`.

```
def better_later_date(date1, date2):
    return <T3>
```

[2 marks]

```
<T3>: later_date(to_iso(date1), to_iso(date2))
```

Question 3: More Datetimes! [12 marks]

Russell sees what Zhu Ming and Brian are working on and decided to take things up a notch. Let's try to sort these dates!

Warning: You should **NOT** use any tuples/lists and built-in functions such as `max`, `min`, `sorted`. You should also **NOT** use any string-related methods such as `.split`, `.find`, `.join`. You may assume that the function `substring(string, start, end, step)` has been provided to you and you should **NOT** use slice operator.

A. Implement a Python function `latest_date` that takes in a comma-separated string-of-dates in the form of **YYYY-MM-DD**. The function should return the latest date among all of the dates.

You may assume that the `later_date(date1, date2)` function from Question 2 has also been provided to you and the dates are all distinct.

Here's a sample execution:

```
>>> latest_date("2024-02-08,2025-10-01,2021-10-13")
"2025-10-01"
>>> latest_date("2024-02-08")
"2024-02-08"
```

Hint: The dates are always of length 10 due to the fixed **YYYY-MM-DD** format.

[3 marks]

```
def latest_date(dates):
    # YYYY-MM-DD is always of length 10
    ans = substring(dates, 0, 10, 1)
    for i in range(0, len(dates), 11):
        ans = later_date(ans, substring(dates, i, i+10, 1))
    return ans
```

B. Implement a Python function `remove_date` that takes in a comma-separated string-of-dates in the form of **YYYY-MM-DD** and a target date in the same form of **YYYY-MM-DD**. The function should return the comma-separated string with the target date removed.

You may assume that the dates are all distinct and the target date exists within the string-of-dates.

Note that you may use the function `latest_date(dates)` from Question 3A.

Here's a sample execution:

```
>>> remove_date("2024-02-08,2025-10-01,2021-10-13", "2025-10-01")
"2024-02-08,2021-10-13"
>>> remove_date("2024-02-08", "2024-02-08")
""
```

[3 marks]

```
def remove_date(dates, target_date):
    ans = ""
    for i in range(0, len(dates), 11):
        date = substring(dates, i, i+10, 1)
        if date != target_date:
            ans += date + ","
    # Remove the last comma
    return substring(ans, 0, len(ans)-1, 1)
```

C. We are finally ready to sort these dates in descending order and impress Russell with your Python skills!

The way we are going to do this is by doing what is called the *selection sort*. In summary, you repeatedly take the largest date from the dates that remain, remove that largest date from it, and put the largest date into your (partially complete but sorted) collection of dates. In the end, your collection of dates is guaranteed to be complete and sorted!

Implement an **ITERATIVE** Python function `sort_dates` that takes in a comma-separated string-of-dates in the form of **YYYY-MM-DD**. The function should return the comma-separated string with all the dates sorted in decreasing order. You may assume there is at least date inside the string and the dates are all distinct. You are strongly encouraged to use the functions defined in Questions 3A and 3B.

Here's a sample execution:

```
>>> sort_dates("2024-02-08,2025-10-01,2021-10-13")
"2025-10-01,2024-02-08,2021-10-13"
>>> sort_dates("2024-02-08")
"2024-02-08"
```

[3 marks]

```
def sort_dates(dates):
    ans = ""
    while dates != "":
        last = latest_date(dates)
        dates = remove_date(dates, last)
        ans += last + ","
    # Remove the last comma
    return substring(ans, 0, len(ans)-1, 1)
```

D. Implement the same function `sort_dates` but **RECURSIVELY**.

[3 marks]

```
def sort_dates(dates):
    if "," not in dates:
        return dates
    else:
        last = latest_date(dates)
        dates = remove_date(dates, last)
        return last + "," + sort_dates(dates)
```

E. **[OPTIONAL]** Is it possible to sort these dates on an ascending order using the same helper functions provided in Questions 3A and 3B?

[0 marks]

Of course you can (with just a small tweak)! Here's how can you do it either iteratively or recursively.

```
def sort_dates(dates):
    ans = ""
    while dates != "":
        last = latest_date(dates)
        dates = remove_date(dates, last)
        ans = last + "," + ans # Just change this line
    return substring(ans, 0, len(ans)-1, 1)

def sort_dates(dates):
    if "," not in dates:
        return dates
    else:
        last = latest_date(dates)
        dates = remove_date(dates, last)
        # Just change this line
        return sort_dates(dates) + "," + last
```

Question 4: Burger Ming [23 marks]

INSTRUCTIONS: Please read the question description clearly before you attempt this problem! You are NOT allowed to use TUPLES and any Python data types which have not yet been taught in class. You should NOT use any string-related methods such as `.split`, `.find`, `.join`. You may assume that the function `substring(string, start, end, step)` has been provided to you and you should NOT use slice operator.

Zhu Ming wants to run a business when he's not teaching his CS1010S class. He asked Russell, who loves all things Python, to work alongside him on creating a burger restaurant called Burger Ming!

As a CEO, Zhu Ming tasked Russell to build a system that represents the orders from the prospective customers of Burger Ming using Python. However, Russell realized that this is not an easy task to do all by himself. Therefore, knowing that you are currently taking CS1010S, he asked for your help.

To start off, Russell explains his Big Brain idea to you, which involves abstracting burger information into prime number! The rationale is inspired by the *The Fundamental Theorem of Arithmetic* (also known as *Unique Prime Factorization Theorem*), which states that any integer number greater than 1 can be uniquely represented as a product of prime numbers. For instance, the number 60 can be represented as $2^2 \cdot 3^1 \cdot 5^1$.

Following this idea, Russell proposes that the **four** types of burgers in Burger Ming can be represented/encoded by the following prime numbers:

- `BigMing`: 2
- `HamBurger`: 3
- `CheeseBurger`: 5
- `HuatBurger`: 7

The `Order` can therefore be represented as an integer where the exponents in the prime factorization indicate the quantity of each type of burger ordered. For example, the `Order` of 2 `BigMing`, 1 `HamBurger`, 1 `CheeseBurger` and 0 `HuatBurger` can be represented by the integer number 60 = $2^2 \cdot 3^1 \cdot 5^1 \cdot 7^0$. Notice that the exponents are the quantities of each type of burger.

Knowing that this is a challenging tasks, Russell has provided you an abstraction `Burger` that supports **only** three functions:

- `make_burger(name)` that takes in an input name (a string) and returns a new burger. There are only **FOUR (4)** possible names for the burger: `"BigMing"`, `"HamBurger"`, `"CheeseBurger"`, `"HuatBurger"`
- `get_burger_name(burger)` that takes in a burger (a `Burger`) and returns the name of the burger (a string).
- `get_prime_encoder(burger)` that takes in a burger (a `Burger`) and returns corresponding `Burger` prime representation (an integer).

Your task is to implement the abstraction `Order` that supports these functions:

- `make_empty_order()` that takes in no inputs and returns a new order.
- `add_to_order(order, burger)` that takes in an order (an `Order`) and a burger (a `Burger`) and adds a burger into the order.
- `remove_from_order(order, burger)` that takes in an order (an `Order`) and a burger (a `Burger`) and removes **one** occurences of the burger from the order.
- `combine_order(order1, order2)` that takes in two orders, combines both orders into a single order, and return that one order.
- `is_equal_order(order1, order2)` that takes in two orders, and checks if they are both equivalent orders, i.e. 1 `BigMing` and 2 `CheeseBurger` is equivalent to 2 `CheeseBurger` and 1 `BigMing`.
- `count_burger(order, burger)` that takes in an order (an `Order`) and a burger (a `Burger`) and returns the number of occurences of the burger in the order.

[IMPORTANT!] For the remaining parts of this question, you should not break the abstraction of `Burger` in your code.

A. Implement `make_empty_order` without using tuples. [1 mark]

```
def make_empty_order():
    return 1
```

B. Implement `add_to_order` and `remove_from_order`. [5 marks]

```
def add_to_order(order, burger):
    return order * get_prime_encoder(burger)

def remove_from_order(order, burger):
    if order == 1:
        return 1
    multiplier = get_prime_encoder(burger)
    if order % multiplier == 0:
        return order // multiplier
    return order
```

C. Obviously the function `add_to_order` only handles an addition of one burger. We have not considered the case of bulk ordering. Implement the function `bulk_add_to_order` that takes in an order, a burger (a `Burger`), and the quantity. It returns the modified order after adding such quantity of the given burger. [2 marks]

```
def bulk_add_to_order(order, burger, n):
    for i in range(0, n, 1):
        order = add_to_order(order, burger)
    return order
```

Common pitfalls include not reassigning the new order to the current order.

D. Implement `combine_order`. [3 marks]

```
def combine_order(order1, order2):
    return order1 * order2
```

Note that storing the burgers as prime numbers instead of strings makes this part of the job much easier.

E. Implement `is_equal_order`.

```
>>> Big_Ming = make_burger("BigMing")
>>> Cheese_Burger = make_burger("CheeseBurger")
>>> prime_meal = make_empty_order()
>>> prime_meal = add_to_order(prime_meal, Big_Ming)
>>> prime_meal = add_to_order(prime_meal, Big_Ming)
>>> prime_meal = add_to_order(prime_meal, Cheese_Burger)
>>> emirp_meal = make_empty_order()
>>> emirp_meal = add_to_order(emirp_meal, Cheese_Burger)
>>> emirp_meal = add_to_order(emirp_meal, Big_Ming)
>>> emirp_meal = add_to_order(emirp_meal, Big_Ming)
>>> is_equal_order(prime_meal, emirp_meal)
True # 2 BigMing and 1 CheeseBurger == 1 CheeseBurger and 2 BigMing

>>> diff_meal = make_empty_order()
>>> diff_meal = add_to_order(diff_meal, Big_Ming)
>>> diff_meal = add_to_order(diff_meal, Big_Ming)
>>> is_equal_order(prime_meal, diff_meal)
False # 2 BigMing and 1 CheeseBurger != 2 BigMing
```

[3 marks]

```
def is_equal_order(order1, order2):
    return order1 == order2
```

Note that storing the burgers as prime numbers instead of strings makes this part of the job much easier.

F. Implement `count_burger`. [3 marks]

```
def count_burger(order, burger):
    ans = 0
    while True:
        new_order = remove_from_order(order, burger)
        if is_equal_order(order, new_order):
            break
        else:
            order = new_order
            ans += 1
    return ans
```

Russell initially implemented a simple and hardcoded version of `get_prime_encoder(burger)` as follows:

```
def get_prime_encoder(burger):
    if get_burger_name(burger) == "BigMing":
        return 2
    elif get_burger_name(burger) == "HamBurger":
        return 3
    elif get_burger_name(burger) == "CheeseBurger":
        return 5
    elif get_burger_name(burger) == "HuatBurger":
        return 7
```

However, as the business of Burger Ming continues to thrive and expand, it becomes impractical to hardcode `get_prime_encoder` for an arbitrary number of burger types. Suppose we modify the implementation of `Burger` such that we do not know how many different burger names are there and the `get_burger_name(burger)` function is replaced by `get_burger_id(burger)`. Now, `Burger` has the following functions:

- `make_burger(name)` that takes in an input name (a string) and returns a new burger.
- `get_burger_id(burger)` that takes in a burger (a `Burger`) and returns a single integer (an `int`) between 1 to the number of distinct burger names, which is unknown.

A sample execution is shown below:

```
>>> prosperity = make_burger("LuckyBurger")
>>> get_burger_id(prosperity)
161     # wow Zhu Ming sells a lot of burgers
>>> Big_Ming = make_burger("BigMing")
>>> get_burger_id(Big_Ming)
1       # first burger he came up with
```

G. Modify the function `get_prime_encoder` such that it addresses the implementation changes stated above. You are not suppose to change your previous `Order` implementations, but if you truly embrace the idea of abstraction barrier, the modifications should be confined within the `get_prime_encoder` function. A sample execution is shown below:

```
>>> o1 = make_empty_order()
>>> o1 = add_to_order(o1, prosperity)
>>> o2 = make_empty_order()
>>> o2 = add_to_order(o2, prosperity)
>>> is_equal_order(o1, o2)
True

>>> o3 = make_empty_order()
>>> o3 = add_to_order(o3, chocolate)
>>> is_equal_order(o1, o3)
False

>>> o3 = add_to_order(o3, prosperity)
>>> o3 = remove_from_order(o3, prosperity)
>>> o4 = make_empty_order()
>>> o4 = add_to_order(o4, chocolate)
>>> is_equal_order(o3, o4)
True

>>> o5 = combine_order(o1, o3)
>>> o6 = combine_order(o2, o4)
>>> is_equal_order(o5, o6)
True
```

[6 marks]

The only change is on `get_prime_encoder` since we have already sticked to the prime number implementation.

```
def next_prime(prime):
    while True:
        prime += 1
        if is_prime(prime): # is_prime is given in Appendix
            return prime
```

```
def nth_prime(n): # helper function to get the nth prime number
    prime = 2
    while n != 1:
        prime = next_prime(prime)
        n -= 1
    return prime

def get_prime_encoder(burger):
    # Find the burger_id-th prime number
    burger_id = get_burger_id(burger)
    return nth_prime(burger_id)
```

For your information, it is possible to solve this question using string, or tuple if the question allowed. However, we choose to use prime numbers implementation simply because it is simple and elegant. This uniqueness property simplifies the logic of the function and makes it easier to implement, as there is no risk of collisions or ambiguity when determining the burger order combination. Therefore, knowing the prime numbers idea offer a straightforward solution for providing unique identifiers for burger types.

Appendix

The following are some functions that were introduced in class. For your reference, they are reproduced here.

```
def substring(s, start, end, step):
    res = ""
    while start < len(s) and start < end:
        res += s[start]
        start += step
    return res

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

def factorial_iter(n):
    result = 1
    for i in range(2, n+1, 1):
        result *= i
    return result

def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

def is_prime(n):
    if n < 2:
        return False
    elif n <= 3:
        return True
    else:
        for d in range(2, int(n**0.5)+1):
            if n % d == 0:
                return False
        return True
```

Scratch Paper

— E N D O F P A P E R —