**NATIONAL UNIVERSITY OF SINGAPORE**

Department of Computer Science, School of Computing

**IT5001—Software Development Fundamentals**

Academic Year 2023/2024, Semester 2

**PROBLEM SET 2**

# CONTROL STRUCTURES, RECURSION & INDUCTIVE REASONING

These exercises are optional and ungraded, and for your own practice only. Contact `yongqi@nus.edu.sg` for queries.

## SIMPLE EXERCISES

**Question 1**. The following program fragment determines the maximum of three numbers.

```
if a >= b and a >= c
  max = a
if b >= a and b >= c
  max = b
if c >= a and c >= b
  max = c
```

For each of the code fragments below, fill in the missing parts . . . so that it can be used in place of the selection constructs in the program above, to effectively output the maximum of three input variables.

```
if ...:
    max = a
else:
    if ...:
        max = b
    else:
        max = c
```

```
max = a
if ...:
    max = b
if ...:
    max = c
```

```
if a >= b:
    if ...:
        max = a
    else:
        max = ...
else:
    if ...:
        max = b
    else:
        max = ...
```

**Question 2**. What differences are there in the following three functions?

```
def foo1():
    if True:
        if False:
            print(1)
        else:
            print(2)

def foo2():
    if True:
        if False:
            print(1)
    else:
        print(2)

def foo3():
    if False:
        if True:
            print(1)
        else:
            print(2)
```

## SHORT PROGRAMMING QUESTIONS

**Question 3**. John wrote the following program to print all the numbers from 1 to 10:

```
for i in range(1, 11):
    print(i)
```

Jane argues that she can use a while loop to do the exact same. Is that true? What would Jane's program look like?

**Question 4**. Deduce the functionality of the following program fragment and re-compose the nested if-else statements into a form that is easier to understand.

```
if a > 0:
    if a >= 1000:
        a = 0
    else:
        if a < 500:
            a *= 2
        else:
            a *= 10
else:
    a += 3
```

**Question 5**. You are writing a function to determine the cost of a ride by either Grab or Gojek. The cost of a ride is determined by

$$\mathit{flagDownFare} + \mathit{distance} \times \mathit{perKmRate}$$

For Grab, the *flagDownFare* is $3 and the *perKmRate* is $0.50. For Gojek, the *flagDownFare* is $2.50 and the *perKmRate* is $0.60.

Write `cab_fare(ride, distance)` where ride is either `'Grab'` or `'Gojek'` and `distance` is the distance of the ride in kilometres. Your function should return the cost of the ride in cents.

**Question 6**. Nowadays, we can customize the food that we order. For example, you can order your burger with extra or no cheese. In this exercise, write a function that takes a string as the customization and compute the price for burgers with the code names for the customization. You are given the price list for ingredients:

| **Ingredient** | **Price** |
|---|---|
| B for bun | $0.50 |
| C for cheese | $0.80 |
| P for patty | $1.50 |
| V for veggies | $0.70 |
| O for onions | $0.40 |
| M for mushrooms | $0.90 |

Write a function `burger_price(burger)` that takes in a burger and returns the price of the burger.

An example run follows.

```
>>> burger_price('BVPB')
3.2
```

**Question 7**. A car requires 1m of parking space, while a bus requires 2m. Write a function `parking` that takes in the amount of parking space available, and returns the number of ways you can park vehicles in that parking space. For example, `parking(3)` should reeturn `3` because you can do car-car-car, car-bus or bus-car. Do so recursively.

**Question 8**. $\pi$ can be approximated by

$$\pi \approx \sum_{i=0}^{\infty} \frac{4 \times (-1)^i}{2i+1}$$

Let $\pi_i$ be the sum of the first $i+1$ terms of the above summation for all $i \in \mathbb{Z}_{\geq 0}$. Write a function `approx_pi(delta)` that takes in some $\delta$ and returns the first $\pi_i$ such that $|\pi_i - \pi_{i-1}| \leq |\delta|$.

**Question 9**. Write a function `sum_digits(n)` that sums the digits of a nonnegative integer $n$. Do so using a `while` loop, a `for` loop, and recursively. For your recursive function, you may choose to write a proof by induction that it works. Adopt offensive programming strategies.

**Question 10**. Write a function `all_odd(n)` that checks if some nonnegative integer $n$ contains only odd digits. Do so using a `while` loop, a `for` loop, and recursively. For your recursive function, you may choose to write a proof by induction that it works. Adopt offensive programming strategies.

**Question 11**. The *fibonacci* series is defined as:

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 \\ f_2 &= f_1 + f_0 = 2 \\ f_3 &= f_2 + f_1 = 3 \\ &\cdots \end{aligned}$$

or as a recurrence relation,

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

Write a function `fibonacci(n)` to determine the $n^{\text{th}}$ fibonacci number. Do so iteratively and recursively. Adopt offensive programming strategies.

**Question 12**. Write a function `can_be_float(s)` that takes some $s$ and returns `True` if $s$ can be converted into a float.

**Question 13**. Based on the Taylor Series for the following:

$$\begin{aligned} \arcsin x &= \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} \qquad = x + \frac{x^3}{6} + \frac{3x^5}{40} + \ldots \qquad \text{for } |x| \le 1 \\ \arccos x &= \frac{\pi}{2} - \arcsin x \\ &= \frac{\pi}{2} - \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} = \frac{\pi}{2} - x - \frac{x^3}{6} - \frac{3x^5}{40} - \ldots \quad \text{for } |x| \le 1 \end{aligned}$$

Write functions `arcsin(x, no_terms)` and `arccos(x, no_terms)` that estimate the above up to `no_terms` terms of the summation. Try doing so iteratively, and then recursively.

## LONG PROGRAMMING QUESTION

**Question 14**. Let's create a guessing game! Write a function `game` that first generates a random number between 1 and 100 (both inclusive), then allows the user to make a guess. If the guess is correct, then print a success message, otherwise, print a failure message. Example runs follow.

```
>>> game()
# assume that 39 was the generated random number
Make a guess: 12
Wrong!
>>> game()
# assume that 50 was the generated random number
Make a guess: 50
Correct!
```

**Question 15**. Now, allow the user to continue guessing a number until they get the right number. If their guess is wrong, pring a message saying whether their guess is smaller or bigger than the correct number. Example runs follow.

```
>>> game()
# assume that 39 was the generated random number
Make a guess: 50
Too big
Make a guess: 25
Too small
Make a guess: 38
Too small
Make a guess: 44
Too big
Make a guess: 41
Too big
Make a guess: 40
Too big
Make a guess: 39
Correct!
```

**Question 16**. A valid guess is an integer between 1 and 100 inclusive. Now amend your function such that it only allows the user to make up to three valid guesses. Defensively allow the user to enter non-integer values which are not considered as valid guesses. Example runs follow.

```
>>> game()
# assume that 39 was the generated random number
Make a guess: 50
Too big
Make a guess: -1
Make a guess: 38
Too small
Make a guess: lolol
Make a guess: 101
Make a guess: 41
Too big
The answer was 39!
```

# SOLUTIONS

**Question 1**. Solution is self-explanatory.

```python
if a >= b and a >= c:
    max = a
else:
    if b >= c:
        max = b
    else:
        max = c
```

```python
max = a
if b >= a:
    max = b
if c >= b:
    max = c
```

```python
if a >= b:
    if a >= c:
        max = a
    else:
        max = c
else:
    if b >= c:
        max = b
    else:
        max = c
```

**Question 2**. Note that a clause like `if True` is not asserting that something is `True`, but rather based on the semantics of conditional statements, we are simply evaluating the condition (which is an expression) `True` and if the result is truthy, we execute the block underneath it. Since `True` is always truthy, we are always executing the block underneath the `if` clause. Such code will never be written in real life, but only serves as a toy example to help you understand the semantics of conditional statements:

```python
def f(a):
    if True:
        print(a)
# we should write the following instead since print(a) will always
# be executed
def f(a):
    print(a)
```

Getting back to the question, `foo1` will print 2, `foo2` and `foo3` will print nothing. Indentation is extremely important in Python programming since it denotes the scope that a code block belongs to; for example, in `foo2` the statement `print(2)` does not execute because the `if True` clause already dictates that the `else` clause will not be executed.

**Question 3**. Jane might have used a `while` loop as follows:

```python
i = 1
while i <= 10:
    print(i)
    i += 1
```

Or more generally:

```python
it = iter(range(1, 11))
while True:
    try:
        i = next(it)
        print(i)
    except StopIteration:
        break
```

In fact, we can always simulate a **for** loop using a **while** loop. In programming language terms, **for** loops are syntactic sugar for **while** loops.

**Question 4**. Notice that we set $a$ as 0 immediately if $a \geq 1000$. Otherwise, if $0 < a < 500$, we multiply $a$ by 2. Otherwise, if $500 \leq a < 1000$, we multiply $a$ by 10. Then, if $a \leq 0$, we increase $a$ by 3.

We can swap the order of the second and third condition to get the following:

```python
if 1000 <= a:
    a = 0
elif 500 <= a < 1000:
    a *= 10
elif 0 < a < 500:
    a *= 2
elif a < 0:
    a += 3
```

Notice in the second condition, $a < 1000$ is obviously true because to even get to this point in the program, the previous condition ($a \geq 1000$) must be false. Therefore, we can remove that condition to get

```python
if 1000 <= a:
    a = 0
elif 500 <= a:
    a *= 10
elif 0 < a < 500:
    a *= 2
elif a < 0:
    a += 3
```

Repeat this process to finally get

```python
if 1000 <= a:
    a = 0
elif 500 <= a:
    a *= 10
```

```python
elif 0 < a:
    a *= 2
else:
    a += 3
```

**Question 5**. Solution is self-explanatory.

```python
def cab_fare(ride, distance):
    if ride == 'Grab':
        return 300 + int(distance * 50)
    elif ride == 'Gojek':
        return 250 + int(distance * 60)
    raise ValueError('ride must be Grab or Gojek')
```

**Question 6**. There are several ways we can iterate over strings: to iterate over the characters directly (as we have seen in Lecture 3), or to iterate over the indices of characters in the string:

```python
def foo(s):
    for i in range(len(s)):
        # i is the index of the character you want
        char = s[i]
        print(i, char)
```

For this problem we will simply iterate over the characters of the string.

We first need to take a single ingredient and get its price. For this we can write a function `ingredient_price` that does so; since this function will need to return different things depending on the ingredient, we can use conditional statements. The following function definition is self-explanatory.

```python
def ingredient_price(ingredient):
    if ingredient == 'B':
        return 0.5
    if ingredient == 'C':
        return 0.8
    if ingredient == 'P':
        return 1.5
    if ingredient == 'V':
        return 0.7
    if ingredient == 'O':
        return 0.4
    return 0.9 # assume that here ingredient can only be M
```

Once we have this, we can now obtain the price of the entire burger. We can use the accumulator pattern.

```python
def burger_price(burger):
    acc = 0 # empty burger costs $0
    for ingredient in burger: # iterate over characters of string
        # add price to accumulator
        acc += ingredient_price(ingredient)
    return acc # answer is in the accumulator
```

**Question 7**. Suppose in the parking space, the rightmost vehicle is a car. That means there are $parking(n-1)$ configurations where this is possible. Suppose in the parking space, the rightmost vehicle is a bus. That means there are $parking(n-2)$ configurations where this is possible. Therefore, your solution might look something like this.

```python
def parking(n):
    if n < 0:
        return 0
    if n < 2:
        return 1
    return parking(n - 1) + \ # case where rightmost veh is car
           parking(n - 2)     # case where rightmost veh is bus
```

**Question 8**. Since $\pi_i$ is the sum of the first $i+1$ terms of the summation, we can use the accumulator pattern. Notice that $\pi_i - \pi_{i-1}$ is the $(i+1)^{\text{th}}$ term in the series. Then, the solutions become self-explanatory.

```python
def approx_pi(delta):
    # make sure delta is positive
    delta = abs(delta)
    # pi starts at 0
    current_pi: float = 0
    # first term is 4
    current_term: float = 4
    # i is the `i` in the term
    i: int = 0
    while abs(current_term) > delta:
        # get the current term
        current_term = 4 * (-1) ** i / (2 * i + 1)
        # add to pi
        current_pi += current_term
        # increase i for the next term
        i += 1
    return current_pi
```

**Question 9.** Two main ideas: 1) get the rightmost digit of $n$ by doing $n \mod 10$, and get the remaining digits of $n$ by doing $\lfloor n/10 \rfloor$.

```
def sum_digits(n):
    if not isinstance(n, int):
        raise TypeError('n must be int')
    if n < 0:
        raise ValueError('n must be nonnegative')
    result: int = 0
    while n > 0: # repeat the following until there is nothing to sum
        result += n % 10 # n % 10 gets the rightmost digit
        n //= 10 # remove the rightmost digit
    return result
```

To use a **for** loop, we can convert $n$ into a string, then as we iterate through the characters of the string, convert it back to an int before we add to the result in the accumulator pattern.

```
def sum_digits(n):
    if not isinstance(n, int):
        raise TypeError('n must be int')
    if n < 0:
        raise ValueError('n must be nonnegative')
    result: int = 0
    for digit in str(n):
        result += int(digit)
    return result
```

To so recursively, we can let the base case be 0, where we return 0. Suppose then we are given some number, say, 12345. Let us try to reduce it to a smaller problem, e.g. 1234. Suppose `sum_digit` correctly returns $1+2+3+4$. Then, clearly, `sum_digit(12345)` is equal to $1+2+3+4+5$ which can be computed `5 + sum_digit(1234)`. More generally, `sum_digit` of a number is its rightmost digit + `sum_digit` of the remaining digits. Thus, we can define the following function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ n \mod 10 + f(\lfloor n/10 \rfloor) & \text{otherwise} \end{cases}$$

Proof: Basis. $f(0) = 0$. $f(d) = d + f(0) = d + 0 = d$ for some single-digit number $d$. Trivial.
Inductive. Suppose $f(d_0d_1d_2 \ldots d_n) = \sum_{i=0}^{n} d_i$ where each $d_i$ is a digit of some nonnegative number. Then,

$$\begin{aligned} f(d_0d_1d_2 \ldots d_nd_{n+1}) &= d_{n+1} + f(d_0d_1d_2 \ldots d_n) \\ &= d_{n+1} + \sum_{i=0}^{n} d_i \\ &= \sum_{i=0}^{n+1} d_i \end{aligned}$$

Writing code for the above is trivial.

```python
def sum_digits(n):
    if not isinstance(n, int):
        raise TypeError('n must be int')
    if n < 0:
        raise ValueError('n must be nonnegative')
    return 0 if n == 0 else \
           n % 10 + sum_digit(n // 10)
```

**Question 10**. Loop through each digit, and return early when you see an even digit. If after looping through all digits you are unable to find an even digit, then they are all odd.

```python
def all_odd(n):
    if not isinstance(n, int):
        raise TypeError('n must be int')
    if n < 0:
        raise ValueError('n must be nonnegative')
    while n >= 10: # while n is not a single digit number
        if n % 2 == 0: # equivalent to (n % 10) % 2 == 0
            return False
        n //= 10
    return n % 2 == 1
```

```python
def all_odd(n):
    if not isinstance(n, int):
        raise TypeError('n must be int')
    if n < 0:
        raise ValueError('n must be nonnegative')
    for digit in str(n):
        if int(digit) % 2 == 0:
            return False
    return True
```

To do so recursively, suppose we define a function $odd(d)$ that returns true if $d$ is odd. One base case is a single digit number, where `all_odd` of a single digit number $d$ is simply $odd(d)$. For the recursive case, we know that any number has all odd digits when the rightmost digit is odd and the remaining digits have all odd digits as well. So, we can define the following function:

$$f(n) = \begin{cases} odd(n) & \text{if } 0 \le n < 10 \\ odd(n \mod 10) \texttt{ and } f(\lfloor n/10 \rfloor) & \text{otherwise} \end{cases}$$

Proof: Basis. $f(d) = odd(d)$ for some single-digit number $d$. Trivial.
Inductive. Suppose $f(d_0 d_1 d_2 \dots d_n) = \bigwedge_{i=0}^{n} odd(d_i)$ where each $d_i$ is a digit of some nonnegative number

($\sum$ denotes summation, $\bigwedge$ denotes and on all of the elements). Then,

$$\begin{aligned} f(d_0d_1d_2\ldots d_nd_{n+1}) &= odd(d_{n+1}) \text{ and } f(d_0d_1d_2\ldots d_n) \\ &= odd(d_{n+1}) \text{ and } \bigwedge_{i=0}^{n} odd(d_i) \\ &= \bigwedge_{i=0}^{n+1} odd(d_i) \end{aligned}$$

Therefore we get

```python
def all_odd(n):
    if not isinstance(n, int):
        raise TypeError('n must be int')
    if n < 0:
        raise ValueError('n must be nonnegative')
    return n % 2 == 1 if 0 <= n < 10 else \
           (n % 2 == 1) and all_odd(n // 10)
```

**Question 11**. The recursive case is simple:

```python
def fibonacci(n):
    if n < 0:
        raise ValueError('n must be nonnegative')
    return 1 if n <= 1 else \
           fibonacci(n - 1) + fibonacci(n - 2)
```

Using loops, the idea is to maintain $\mathit{fibonacci}(i)$ and $\mathit{fibonacci}(i+1)$ as variables, get $\mathit{fibonacci}(i+2)$ by taking the sum, and then updating $\mathit{fibonacci}(i)$ and $\mathit{fibonacci}(i+1)$ as we calculate the next fibonacci number. Using either while or for loops is fine.

```python
def fibonacci(n):
    if n < 0:
        raise ValueError('n must be nonnegative')
    f_0 = 1
    f_1 = 1
    result = 0
    for i in range(2, n + 1):
        result = f_0 + f_1
        f_0 = f_1
        f_1 = result
    return f_1
```

**Question 12**. Simple, we can use `float` to do so. We try converting $s$ into a float, and if it works, we return true; if we get an exception, we return false.

```python
def can_be_float(s):
    try:
        float(s)
        return True
    except:
        return False
```

**Question 13**. We first work on arcsin since arccos depends on arcsin. We can first write a function to obtain the $i^{\text{th}}$ term in the Taylor Series expansion of arcsin:

```python
from math import factorial

def sin_term(x, i):
    return factorial(2 * i) * x ** (2 * i + 1) / \
        (4 ** i * factorial(i) ** 2 * (2 * i + 1))
```

For brevity we now express the `sin_term(x, i)` function as $s(x, i)$, therefore

$$\arcsin x = \sum_{i=0}^{\infty} s(x, i)$$

Computing arcsin iteratively is straightforward using the accumulator pattern:

```python
def arcsin(x, no_terms):
    acc = 0
    for i in range(no_terms):
        acc += sin_term(x, i)
    return acc
```

Doing so recursively is relatively straightforward as well. Observe the following expansion:

$$\begin{aligned}\arcsin(x, n) &= \sum_{i=0}^{n-1} s(x, i)\\ &= s(x, n-1) + \sum_{i=0}^{n-2} s(x, i)\\ &= s(x, n-1) + \arcsin(x, n-1)\end{aligned}$$

We can therefore define arcsin with the following recurrence relation:

$$\arcsin(x, n) = \begin{cases} 0 & \text{if } n \leq 0 \\ s(x, n-1) + \arcsin(x, n-1) & \text{otherwise} \end{cases}$$

Writing code for this is trivial.

```python
def arcsin(x, no_terms):
    if no_terms <= 0:
        return 0
    return sin_term(x, no_terms - 1) + arcsin(x, no_terms - 1)
```

Then defining arccos becomes a trivial task.

```python
from math import pi
def arccos(x, no_terms):
    return pi / 2 - arcsin(x, no_terms)
```

**Long Programming Question. Question 14**. Pretty straightforward:

```python
from random import randint
def game():
    # generate the correct answer
    correct_answer = randint(1, 100)
    # get guess from user
    guess = int(input('Make a guess: '))
    # print the right message depending on the guess
    if guess == correct_answer:
        print('Correct!')
    else:
        print('Wrong!')
```

**Question 15.** Similar to the lecture example, we can use a **while** loop to repeatedly get guesses from the user until they make the right guess. This time, we also need to print the right message depending on whether their guess is too big or too small.

```python
from random import randint
def game():
    correct_answer = randint(1, 100)
    guess = -1 # just set to some incorrect value first
    while guess != correct_answer:
        # get guess from user
        guess = int(input('Make a guess: '))
        # print the right message depending on the guess
        if guess > correct_answer:
            print('Too big')
        elif guess < correct_answer: # can use if instead of elif here
            print('Too small')
        # nothing to do if guess == correct_answer since we
        # will break out of the loop and print Correct!
    # at this point, guess == correct_answer
    print('Correct!')
```

**Question 16**. Now we need to handle the case where the guess is invalid where it is either it is not even an integer (we have to handle this exception) or it is a guess outside of `range(1, 101)`.

```python
from random import randint
def game():
    correct_answer = randint(1, 100)
    # keep track of number of valid attempts left
    no_of_valid_attempts_left = 3
    # repeat until no more valid attempts left
    while no_of_valid_attempts_left > 0:
        # read the guess
        guess = input('Make a guess: ')
        try:
            # attempt to convert to an int.
            guess = int(guess)
        except:
            # here, int conversion failed, skip this iteration so
            # we can read another guess immediately
            continue
        # from the guard clause above, we can assume guess is an int
        # check if the guess is between 1 and 100; if it is not,
        # likewise skip this iteration so we can read another
        # guess immediately
        if guess < 1 or guess > 100:
            continue
        # from the guard clauses above, we can assume
        # the guess is valid
        # decrease the number of valid attempts
        no_of_valid_attempts_left -= 1
        # if the guess is correct, print Correct! and terminate
        # the loop
        if guess == correct_answer:
            print('Correct!')
            break
        if guess > correct_answer:
            print('Too big')
        else:
            print('Too small')
    print(f'The answer was {correct_answer}!')
```

**– End of Problem Set 2 –**