FIT1045/53 Workbook

June 19, 2020

Contents

Preface

This workbook is an additional learning resource for FIT1045 and FIT1053. This is our first semester offering a workbook, so the workbook will grow as the semester progresses. This workbook is not sufficient for the unit. You must also complete lectures, workshops, and tutorials.

The workbook will be useful in tutorials during time dedicated to self-directed-learning; as a tool for identifying any gaps in your knowledge; and for providing materials with which to practice. You can use the workbook as much or as little as is useful for you.

The questions in this workbook reflect the questions you can expect to see on the in-semester tests, and the final exam. The difficulty of the questions can be understood like so:

1. **Clarify** questions allow you to identify any misunderstandings you may have with the fundamental tools you will learn. These are very simple questions. If you can do the practice questions, there is no need to do these questions.

2. **Practice** questions allow you to practice the kinds of questions you will see in your tests and exam. Some will be knowledge-based, some will be problem solving. If you can do all the practice questions, you will be fully prepared for the content of the tests and exam. *Always start with the practice questions, and if you cannot do them switch to clarify; if they are too easy switch to challenge.*

3. **Challenge** questions allow you to practice questions that are *more difficult* than most of the questions you will see in your tests and exam. It can be difficult to problem-solve under test conditions, so practicing on questions that are more difficult than what you might expect to see can be useful. Some questions on the final exam will be similar to challenge level questions. Solutions will not be provided for Challenege questions.

4. **Extension** questions are questions that contain *non-assessable* content. They may contain a taster of material that will be taught in later units, or they may show you something cool you can do with the tool you are being taught.

If you are ever confused by a workbook question, feel free to ask about it in your tutorial, on Moodle, or at a consult.

Enjoy!

Part I

Programming Fundamentals

1 Expressions

1.1 Numerical Expressions

Clarify

What do each of these numerical expressions yield?

1. `9+5`
2. `7/2`
3. `3**2`
4. `10%3`
5. `4/2`
6. `-9%2`
7. `-10%-4`
8. `8//4`
9. `14//5`
10. `9.5//3`
11. `6//3.0`
12. `9.5%2`
13. `8.0%3`
14. `2**2**3`
15. `4-3*2`
16. `13/(5%3)`

Practice

What do each of these numerical expressions yield? Make sure you understand when the expression yields a *float* and when it yields an *integer*.

1. `10/(5%2)`
2. `4+11%6//2**2`
3. `2**(9%3+2)+12//(5.5-7.5)`
4. `1*2**(3%2+2)+14//5`
5. `3*2**2**(5//7+1)`
6. `3+2*3**(10//5)-7`
7. `7*-4**4`
8. `-8%-6//3`
9. `3*3/-4`
10. `-1//-5*-8`
11. `5**1-4`
12. `-2-7%4`
13. `1+0%7-9%1`
14. `-8*6//2%2+-1`
15. `6*8*4/-6//-4`
16. `8+4+2*-3/-3`
17. `-9+-1%-7--8%-4`
18. `9/1-7//7%9`
19. `4--8*-4-5%-9`
20. `-8/-7*-7//-5*-9`
21. `0-5-4%6//-2`
22. `-6%8+2+-8//-2`
23. `-6/8//8%8/4`
24. `7--4//-4**6*9`
25. `-5+-2**-3**-9**2`
26. `-4**4%-1**2//8`
27. `7**-10//2**-10/6`
28. `-4+2**2**3--9`
29. `-1**7//4*-2**4`
30. `-5+4+-3//-1*2`
31. `-2%4**3-2**9`
32. `-4--2+8**-7*0`

Challenge

What do each of these numerical expressions yield?

1. `int(False or True)*2**2**(14//5*2%3+2)+2`

Answers

Write the expression in the Python shell and compare output.

1.2 Boolean Expressions

When answering the following questions, keep in mind Python operator precedence given in the lectures, or found in the Python documentation.[1] It can be helpful to think of operators with higher precedence as having brackets around them. For example, to think of `False or not True and True` as being `False or ((not True) and True)`.

Also keep in mind the behavior of the boolean operators given in the lectures, or found in the Python documentation.[2]

Finally, be aware of when type coercion occurs.

Clarify

What do each of these function calls return or boolean expressions yield?[3]

1. `bool(3)`
2. `bool('')`
3. `bool([0])`
4. `bool('cat')`
5. `not 'dog'`
6. `not []`

What do each of these boolean expressions yield? Assume each expression is self-contained and identify which will result in a `NameError`.

Boolean operators

1. `True or False`
2. `True and False`
3. `False or False`
4. `not True or True`
5. `not (True or True)`
6. `not not True`

Comparison operators

1. `7 <= 3`
2. `2 > 2`
3. `1 != 0`
4. `10 < 5 < 20`
5. `3 > 2 < 20`
6. `2 == 2 > -4`

Booleans - with short-circuiting

1. `x or True`
2. `True or x`
3. `False and x`
4. `False or x`
5. `True and False or x`
6. `False and True or True or x`

Comparisons - with booleans

1. `True == 1`
2. `False < 3`
3. `False >= 0`
4. `True == 6`
5. `True == (not not 6)`[4]
6. `False == []`

Practice

What do each of these boolean expressions yield? Assume each expression is self-contained and identify which will result in a `NameError`.

1. `'' or not 0`
2. `False or False and True or True or x`
3. `True or False or 2`
4. `5 or True and False`
5. `3>7 or 9>=15 and 4<5 or not 10==4`
6. `not [] and 2<2 and 'cat'or 7>4<12`

[1] Operator precedence at url: `https://docs.python.org/3/reference/expressions.html#operator-precedence`
[2] Boolean Operator behavior at url: `https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not`.
[3] Truth value testing at url: `https://docs.python.org/3/library/stdtypes.html#truth`
[4] Note that `==` has higher precedence than `not`, which means brackets must be used to avoid a `SyntaxError`.

7. `'word' or range(1,2) or -6>=8 or -3<=-3`
8. `-3<=-3 or ''and -4==-1 or -4==-1`
9. `[] and 'word'or range(1,2) and ['']`
10. `-6>=8 and 0 and -3<=-3 or 'word'`
11. `'' and -4==-1 or -5 or range(0)`
12. `2<0 and None and None or 2<0`
13. `not(-4>-1 or not [''] or not 'w'or 'word')`
14. `'0' and 4 or range(0) or -5`
15. `0 and 3==4 or range(0) or ''`
16. `None and ''and 2<0 and range(1,2)`
17. `range(3,1) or None or [] or 'word'or ''`
18. `None and range(1,2) or None or -5 and -4==-1`
19. `2<0 or [''] and [] or range(5,3) and 'word'`
20. `not([''] and -5 or -4==-1) or ''and 'word'`
21. `-3<=-3 and ''or -5 and '0'or range(-3, -9)`
22. `'0' or -4==-1 or not ('word'or 2<0 and None)`
23. `'' or 7 and -4==-1 and -6>=8 or []`
24. `0 and not 'word'or 'string'or '0'and -4==-1`

Answers

Write the expression in the Python shell and compare output.

2 | Control Flow

2.1 Conditionals

Clarify

Code trace

What is the value of x after the code sequence is executed?

1.
```
x = 5
if x < 10:
    x = 'small'
```
'small'

2.
```
x = True
if x:
    x = 'Yay!'
else:
    x = 'No....'
```
'Yay!'

3.
```
x = 'dog'
if x == 'cat':
    x = 'meow'
elif x == 'dog':
    x = 'woof'
else:
    x = 'animal sound'
```
'woof'

4.
```
x = 6
if x > 10:
    x = 'big'
elif x < 5:
    x = 'small'
```
6

5.
```
x = 25
if x < 10:
    x = 'cold'
elif x > 25:
    x = 'hot'
elif x < 25:
    x = 'average'
else:
    x = 'perfect'
```
'perfect'

6.
```
x = -10
if x < 0:
    x = x * -1
```
10

7.
```
a = 5
b = 8
x = 0
if a < b:
    x = b
else:
    x = a
```
8

8.
```
x = 25
if x%2 == 0:
    x = x // 2
if x%3 == 0:
    x = x // 3
if x%5 == 0:
    x = x // 5
```
x= 5

Coding

Write a control sequence that ...

1. sets x to the smaller of two numbers, a and b.

x = a

if x<b

x = b

if x<c

x = c

2. sets `x` to the largest of three numbers, `a`, `b`, and `c`.

3. sets `parity` to `even` or `odd`, based on the value of `num`.

4. sets `opt` to `'different'`, `'pair'`, or `'triple'`, based on the value of `a`, `b`, and `c`.

5. sets `message` to `'Wear {option}'` based on the value of `temp`. The value `option` should be `'a jacket'` if the temperature is cold, and `'sunscreen'` if the temperature is hot. You may decide what temperature is hot, and what is cold.

if a = b = c

'triple'

if a=b or b=c or a=c:

'pair'

Practice

Write a control sequence that ...

a, b, c

if

a+b > c

else

1. sets `shape` to `'triangle'` or `'not triangle'` given the lengths of three lines.[1]

2. sets `type` to `'equilateral'`, `'isosceles'`, or `'scalene'`, given the lengths of the three sides of a triangle, `x`, `y`, and `z`.

3. sets `rounded` to the decimal `num` rounded to the nearest integer. Follow the IEEE rounding rule 'round to nearest, ties to even' (i.e. if rounding $x.5$, round to the nearest even integer).[2]

4. sorts a list of three numbers.[3]

Answers

Clarify

Code trace

1. `x == 'small'`
2. `x == 'Yay!'`
3. `x == 'woof'`
4. `x == 6`
5. `x == 'perfect'`
6. `x == 10`
7. `x == 8`
8. `x == 5`

Coding

1.
```
a = ?
b = ?
x = a
if b < a:
    x = b
```

2.
```
a = ?
b = ?
c = ?
x = a
if x < b:
    x = b
if x < c:
    x = c
```

3.
```
num = ?
parity = None
if num%2 == 0:
    parity = 'even'
else:
    parity = 'odd'
```

[1] For three lines to make a triangle, every line must be shorter than the sum of the remaining two.

[2] For more information on the IEEE rounding rules, see here: `https://en.wikipedia.org/wiki/IEEE_754`

[3] Attempt after completing section Lists.

4.
```
a = ?
b = ?
c = ?
opt = 'different'
if a == b or b == c or a == c:
    opt = 'pair'
if a == b and b == c:
    opt = 'triple'
```

5.
```
#15 or lower is cold; above 15 is hot
temp = ?
message = 'Wear '
if temp <= 15:
    message = message + 'a jacket'
else:
    message = message + 'sunscreen'
```

Practice

Note that there are many potential implementations. The answers given here are just an example of a possible implementation.

2.
```
x = ?
y = ?
z = ?
type = None
if x == y and  y == z:
    type = 'equilateral'
elif x==y or y==z or x==z:
    type = 'isosceles'
else:
    type = 'scalene'
```

4.
```
lst = [?, ?, ?]
a = lst[0]
b = lst[1]
c = lst[2]
#correct position of a
if a > b or a > c:
    if a > b and a > c:
        lst[2] = a
    else:
        lst[1] = a
#correct position of b
if b < a and b < c:
    lst[0] = b
elif:
    if b > a and b > c:
        lst[2] = b
#correct position of c
if c < a or c < b:
    if c < a and c < b:
        lst[0] = c
    else:
        lst[1] = c
```

2.2 Loops

2.2.1 While-Loops

All questions are **Clarify**. For **Practice** and higher, go to the Loops subsection.

Fill in the blanks

Fill in the blanks, _, so that each loop creates a list of the numbers 1 to 10.

1.
```
x = 1
lst = []
while x <= 10:
    lst = lst + [x]
    x = x + 1
```

2.
```
x = 1
lst = []
while True:
    lst = lst + [x]
    x = x + 1
    if x == 11 :
        break
```

3.
```
x = 2
lst = []
while x < 21:
    lst = lst + [x//2]
    x = x + 2
```

4.
```
x = 0
lst = []
while True:
    if x > 9:
        break
    x = x + 1
    lst = lst + [x]
```

Coding

Write a while-loop that ...

1. ... adds all the numbers from 1 to 100 (inclusive) and stores the total in the variable `x`.
2. ... adds all even numbers from 1 to 50 (inclusive) and stores the total in the variable `x`.
3. ... creates the list `lst = [1,2,3,4,5,6,7,8,9]`.
4. ... creates the list `lst = [10, 8, 6, 4, 2, 0]`.
5. ... creates the list `lst = [1,1,1,1,1,1,1,1,1]`. Use the length of the list in the while condition.
6. ... creates the list `lst = [1,1,1,1,1,1,1,1,1]`. Use a counter.
7. ... creates the list `lst = [1,2,3,4,5,6,7,8,9]`. Use a break statement.
8. ... creates the list `lst = [2,2,2,2,2,2,2,2,2]`. Use a break statement.

2.2.2 For-Loops

All questions are **Clarify**. For **Practice** and higher, go to the Loops subsection. This section should be done after you have completed the Range subsection.

Fix the problem

Identify the problem with the given loop and implement a corrected version.

1.
```
#should add the numbers 1 to 10
total = 0
for num in range(2, 10):   range(1, 11)
    total = total + num
```

2.
```
#should create the string n='12345'
n = ''
lst = [1,2,3,4,5]
for i in range(len(lst)):
    n = n + str(i)
```
str(lst[i])

3.
```
#should double each number in lst
lst = [1,2,3,4,5]
for n in lst: range(len(lst))
    n = n*2
  lst[i] = lst[i]*2
```

4.
```
#should replace all 'o's with 'a's
b = 'bonono'
for i in range(1,6,2):
b2=''  b[i] = 'a'
for i in range(len(b)):
    if b[i] == 'o'
        b2 = b2 + 'a'
    else: b2 += b[i]
```

Coding

Write a for-loop that ...

1. ... adds all the numbers from 1 to 100 (inclusive) and stores the total in the variable `x`.
2. ... adds all even numbers from 1 to 50 (inclusive) and stores the total in the variable `x`.
3. ... triples all the numbers in the given list `lst`.
4. ... creates a list containing ten `5`s.
5. ... creates a list of all the even numbers that are greater than 1 and less than 100.
6. ... takes a string `word` and creates a new string identical `word` but with all `'a'`s removed.

2.2.3 Loops

Clarify

Choose

What kind of loop would you use for the following scenarios?

1. Halving a number until it is smaller than 100. while
2. Doubling every number in a list. for
3. Summing an infinite series to estimate an irrational number. while
4. Summing every digit in a string. for
5. Count the frequency of a character in a string. for
6. Asks for input from the user, and keeps asking until input is valid. while

Coding

Write a while-loop with the same behaviour as the given for-loop.

1.
```
total = 0
for num in range(55, 105, 5):
    total = total + num
```

2.
```
x = 100
for num in range(-1, -10, -1):
    x = x + num
```

3.
```
lst = [1, 2, 3, 4, 5, 6]
total = 0
for num in lst:
    total = total + num
```

4.
```
lst = ['a', 'b', 'a', 'a', 'b', 'a']
for i in range(len(lst)):
    if lst[i] == 'b':
        lst[i] = 'aa'
```

for i in range(len(lst))
lst [i]= lst[i]// 2

Write a for-loop with the same behaviour as the given while-loop:

1. ```
   total = 0
   num = 20
   while num < 42:
       total = total + num
       num = num + 2
   ```
   for n in range(20, 41, 2)

2. ```
   total = 0
   num = 25
   while num <= 100:
       total = total + num
       num = num + 25
   ```

3. ```
   lst = [2, 4, 3, 5, 2, 2, 3]
   i = 0
   while i < len(lst):
       lst[i] = lst[i] // 2
       i = i + 1
   ```

4. ```
   lst = ['P', 'y', 't', 'h', 'o', 'n']
   word = ''
   i = 0
   while i < len(lst):
       word = word + lst[i]
       i = i + 1
   ```

Practice

Write loops with the following behaviours:

1. Generates a list containing the first 64 powers of 2.
2. Generates a list containing all the powers of 2 that are less than 1,000,000.
3. Generates a list containing the first 100 numbers of the Fibonacci sequence. I.e. `[0,1,1,2,3,...]`.[4]
4. Determines whether a given list of numbers is in ascending order.
5. Finds the factorial of the given number.[5]
6. Finds the greatest common divisor of two given numbers.
7. Finds the lowest common multiple of two given numbers.
8. Finds the sum of all numbers in a given string. Assume numbers are positive integers. For example, `'4 and 20 blackbirds'` sums to 24.
9. Finds the sum of all digits in a given number.
10. Finds the mean of a list of numbers.
11. Generates the reverse of a given string. Do not use string slicing or the function `reversed()`.
12. Determines whether a given string is a palindrome. Do not use string slicing or the function `reversed()`.
13. Generates a list of all factors of a given number.
14. Generates the complement of a given binary number. For example, the complement of `'110101'` is `'001010'`.

Challenge

Write loops with the following behaviours:

1. Generates a list containing all the perfect numbers between 1 and 1000. I.e. it should return this list: `[6, 28, 496]`. A perfect number is a positive integer that is equal to the sum of its positive divisors, excluding the number itself. For example, the divisors of 6 are 1, 2, 3, and $1 + 2 + 3 = 6$.[6]
2. Generates a list of the first 100 primes.

[4] Read about Fibonacci numbers here: `https://en.wikipedia.org/wiki/Fibonacci_number`
[5] Read about factorials here: https://en.wikipedia.org/wiki/Factorial
[6] Read about perfect numbers here: `https://en.wikipedia.org/wiki/Perfect_number`

3. Checks whether the given number `num` is a happy number. A happy number is a number where repeatedly calculating the sum of the squares of the number's digits leads to 1. For example, if `num=28` then when the loop terminates `is_happy = True`; if `num=22` then when the loop terminates `is_happy = False`.[7]

4. Generates a list containing all permutations of a given string. For example, `'cat'` would generate `['cat', 'cta', 'act', 'atc', 'tca', 'tac']`.

[7] Let s_0 be our number. Let the sum of the squares of the digits of s_0 be s_1. Let the sum of the squares of the digits of s_1 be s_2. Let the sum of the squares of the digits of s_n be s_{n+1}. A number x_0 is happy if there exists an n whereby $x_n = 1$. For instance, 28 is happy as:

$$\begin{aligned} 2^2 + 8^2 =& 68 \\ 6^2 + 8^2 =& 100 \\ 1^2 + 0^2 + 0^2 =& 1 \end{aligned}$$

Whereas 22 is unhappy as:

$$\begin{aligned} 2^2 + 2^2 =& 8 \\ 8^2 =& 16 \end{aligned}$$

... and 16 is a known unhappy number. The following numbers are known unhappy numbers: 0, 4, 16, 20, 37, 42, 58, 89, or 145. Read more about happy numbers here: `https://mathworld.wolfram.com/HappyNumber.html`. Note that the description given here is a limited definition of happy numbers and not accurate if you are a number theory mathematician.

Answers

While-loops

Fill in the blanks

1. ```
   x = 1
   lst = []
   while x <= 10:
       lst = lst + [x]
       x = x + 1
   ```

2. ```
   x = 1
   lst = []
   while True:
       lst = lst + [x]
       x = x + 1
       if x == 10:
           break
   ```

3. ```
   x = 2
   lst = []
   while x < 21:
       lst = lst + [x//2]
       x = x + 2
   ```

4. ```
   x = 0
   lst = []
   while True:
       if x > 9:
           break
       x = x + 1
       lst = lst + [x]
   ```

Coding

2. ```
   x = 0
   num = 1
   while num <= 50:
       if num%2 == 0:
           x = x + num
       num = num + 1
   ```

4. ```
   lst = []
   num = 10
   while num >= 0:
       lst = lst + [num]
       num = num - 2
   ```

6. ```
   lst = []
   counter = 0
   while counter < 9:
       lst = lst + [1]
       counter = counter + 1
   ```

8. ```
   lst = []
   while True:
       if len(lst) == 9:
           break
       lst = lst + [2]
   ```

For-loops

Fix the problem

1. ```
   #should add the numbers 1 to 10
   total = 0
   for num in range(1, 11):
       total = total + num
   ```

2. ```
   #should create the string n='12345'
   n = ''
   lst = [1,2,3,4,5]
   for i in range(len(lst)):
       n = n + str(lst[i])

   #alternative implementation
   n = ''
   lst = [1,2,3,4,5]
   for num in lst:
       n = n + str(num)
   ```

3.
```
#should double each number in lst
lst = [1,2,3,4,5]
for i in range(len(lst)):
    lst[i] = lst[i]*2
```

4.
```
#should replace all 'o's with 'a's
b = 'bonono'
ban = ''
for char in b:
    if char == 'o':
        ban = ban + 'a'
    else:
        ban = ban + char
```

Coding

1.
```
x = 0
for num in range(1, 101):
    x = x + num
```

2.
```
x = 0
for num in range(0, 51, 2):
    x = x + num
```

3.
```
lst = ??
for i in range(len(lst)):
    lst[i] = lst[i]*3
```

4.
```
lst = []
for _ in range(10):
    lst = lst + [5]
```

5.
```
lst = []
for num in range(2, 100, 2):
    lst = lst + [num]
```

6.
```
word = ??
new = ''
for char in word:
    if word != 'a':
        new = new + char
```

Loops

Clarify

1. As there is a condition to be met, should use a while-loop.
2. This can be done with a while-loop or for-loop. A for-loop will likely be simpler to implement.
3. Usually when estimating an irrational number, the intention is to get within a certain level of error. This means there is a condition to be met, so should use a while-loop.
4. This can be done with a while-loop or for-loop. A for-loop will likely be simpler to implement.
5. This can be done with a while-loop or for-loop. A for-loop will likely be simpler to implement.
6. As there is a condition to be met, should use a while-loop.

For-loops to while-loops

1.
```
total = 0
num = 55
while num <= 100:
    total = total + num
    num = num + 5
```

2.
```
x = 100
num = -1
while num > -10:
    x = x + num
    num = num -1
```

3.
```
lst = [1, 2, 3, 4, 5, 6]
total = 0
i = 0
while i < len(lst):
    total = total + lst[i]
    i = i + 1
```

4.
```
lst = ['a', 'b', 'a', 'a', 'b', 'a']
i = 0
while i < len(lst):
    if lst[i] == 'b':
        lst[i] = 'aa'
    i = i + 1
```

While-loops to for-loops

1.
```
total = 0
for num in range(20, 42, 2):
    total = total + num
```

2.
```
total = 0
for num in range(25, 125, 25):
    total = total + num
```

3.
```
lst = [2, 4, 3, 5, 2, 2, 3]
for i in range(len(lst)):
    lst[i] = lst[i] // 2
```

4.
```
lst = ['P', 'y', 't', 'h', 'o', 'n']
word = ''
for letter in lst:
    word = word + letter
```

Practice

Note that there are many potential implementations. The answers given here are just an example of a possible implementation.

2.
```
lst = []
index = 0
while 2**index < 1000000:
    lst = lst + [2**index]
    index = index + 1
```

4.
```
lst = ??
in_order = True
for i in range(len(lst)-1):
    if lst[i] > lst[i+1]:
        in_order = False
```

6.
```
big_num = ??
small_num = ??
while small_num != 0:
    remainder = big_num % small_num
    big_num = small_num
    small_num = remainder
gcd = big_num
```

8.
```
string = ??
numbers = \
  ['0','1','2','3','4',
  '5','6','7','8','9']
i = 0
sum = 0
while i < len(string):
    if string[i] in numbers:
        j = i
        while j < len(string) and \
    string[j] in numbers:
            j = j+1
        num = string[i:j]
        sum = sum + int(num)
        i = j
    i = i+1
```

10.
```
sum = 0
for num in lst:
    sum = sum + num
mean = sum/len(lst)
```

12.
```
string = ??
palindrome = True
for i in range(len(string)//2):
    if string[i] != string[-(i+1)]:
        palindrome = False
```

14.
```
binary = ??
complement = ''
for digit in binary:
    if digit == '0':
        complement = complement + '1'
    else:
        complement = complement + '0'
```

3 Sequences

Summary of sequence operations:

Operator/Function	Result	Type
`x in seq`	whether `x` is *contained* in `seq`	Boolean
`x not in seq`	whether `x` is *not contained* in `seq`	Boolean
`len(seq)`	number of elements contained in `seq`	integer
`seq[i]`	the ith element of `seq`	varies
`seq[a:b]`	the sub-sequence of `seq` from the ath element (inclusive) to the bth element (exclusive)	same as `seq`

3.1 Strings

Summary of useful string methods:[1]

Operation	Description
`str.count(sub)`	Returns the number of times substring `sub` is in string `str`
`str.find(sub)`	Returns the lowest index `sub` is in `str`, or -1 if not in `str`
`str.format(*args)`	Formats the string `str`
`str.index(sub)`	Returns the lowest index `sub` is in `str`, throws error if not in `str`
`str.join(iter)`	Returns a string which is the concatenation of the strings in `iter`. Strings in `iter` are joined by `str`.
`str.lower()`	Returns a copy of `str` with all cased characters in lower case
`str.split(sep)`	Returns a list of the words in `str`, using `sep` as the delimiter string
`str.strip()`	Removes all white space from beginning and end of `str`
`str.upper()`	Returns a copy of `str` with all cased characters in upper case

Clarify

Basic Operators

What do each of these expressions yield? Ensure you include all information.

'2'

1. `'I have '+ str(2) + 'cats.'`
2. `len('animals')`
3. `'ing' in 'Coding is fun!'` True
4. `'Coding is fun!'[0]`
5. `'Coding is fun!'[10:13]`
6. `len('Python')`
7. `'FIT1045'[:3]`
8. `'Hello, world.'[3:5] + 'magic'[-3:len('magic')]`

[1] For all string methods, see here: `https://docs.python.org/3.8/library/stdtypes.html#string-methods`

String Methods

What do each of these expressions yield? Ensure you include all information.

1. `'antelopes, ants, and alpacas are all animals'.count('an')`
2. `'Python'.find('y')` 1
3. `'Hello'.find('y')` -1
4. `' and '.join(['lions','tigers','bears'])`
5. `'Learn Python in FIT1045'.lower()`
6. `'Do you know methods?'.split()`
7. `'1+3+5=2'.split('+')`
8. `' Clean up... '.strip()`

Practice

Evaluate

What do each of these expressions yield? Ensure you include all information. Which expressions throw errors or do not behave as they ought, and how can they be fixed?

1. `'One wonders on a pond'.lower().count('on')`
2. `len('FIT1045,FIT1008,FIT2004,FIT3155'.split(',')[2])`
3. `'and'.join('7 + 9 + 4'.split('+'))`
4. `'oar'.upper().join('fat rat sat with bat and goat at math'.split('at')[2:5])`
5. `'{} {}s'.format('Emit'.lower()[::-1], 'Drawer'.lower()[::-1])`
6. `'na'*8 + 'batman'.upper()`
7. `'This question is number {}'.format('5 + 2'.split()[0] + '5 + 2'.split()[-1])`
8. `'There are '+ 3+7 + 'movies with Ironman.'`

Code

In the following questions the only strings you can use are those you are given. Write an expression that ...

1. ... evaluates to `'python'` given the string `string = 'Students in FIT1045 code in Python.'`
2. ... evaluates to `'saidallinonebreath'` given the string `string = 'Said all in one breath!'`
3. ... replaces all `e`s in a string with `o`s, given `e='e'` and `o='o'`. Assume the string is `string`.
4. ... evaluates to the given string, but with the words in reverse order. Assume the string is `string`. E.g. `'This is weird.'` becomes `'weird. is This'`

Answers

Write the expression in the Python shell and compare output.

3.2 Lists

Summary of useful string methods:[2]

Operation	Description
`lst.append(x)`	Appends item `x` to list `lst`
`lst.extend(iter)`	Extends list `lst` by adding all items from iterable `iter`
`lst.count(x)`	Returns the number of times item `x` is in list `lst`
`lst.insert(k,x)`	Inserts item `x` at index `k` in list `lst`
`lst.pop()`	Returns the last item in list `lst` and removes it from the list
`lst.remove(x)`	Removes the first occurrence of item `x` in list `lst`
`lst.reverse()`	Reverses all the items in list `lst`

Clarify

Basic Operators

What do each of these expressions yield? Ensure you include all information.

1. `` `I have '+ 2 + ` cats' ``

List Methods

What do each of these expressions yield? Ensure you include all information.

Practice

Evaluate

What do each of these expressions yield? Ensure you include all information. Which expressions throw errors or do not behave as they ought, and how can they be fixed?

1. `` `I have '+ 2 + ` cats' ``

Code

Answers

Write the expression in the Python shell and compare output.

[2] For all string methods, see here: `https://docs.python.org/3.8/library/stdtypes.html#string-methods`

3.3 Range

Clarify

Evaluate

What sequence of numbers is equivalent to the given ranges?

1. `range(6)`
2. `range(2,5)`
3. `range(4,10)`
4. `range(-3,5)`
5. `range(4,2)`
6. `range(2,8,2)`
7. `range(1,10,2)`
8. `range(25,5,-3)`
9. `range(-2,-20,-4)`
10. `range(10,-10,-5)`
11. `range(-14,-2,-3)`
12. `range(-4,15,6)`

Coding

Express each of the following as a range.

1. `[11,12,13,14,15]` 11, 16
2. `[9,8,7,6,5,4]` 9, 3, -1
3. `[-2,-4,-6,-8,-10]` -2, -11, -2
4. `[-5,-4,-3,-2,-1,0,1,2,3,4,5]` -5, 6, 1
5. `[0,1,2,3,4,5,6,7,8,9]`
6. `[0,10,20,30,40,50,60,70,80]`
7. `[0,-4,-8,-12,-16,-20,-24]`
8. `-15,-10,-5,0,5,10,15,20`

Practice

What do each of these expressions yield?

1. `len(range(2,8))`
2. `3 in range(-2,5)`
3. `range(-7,-2)[4]`
4. `range(4,20)[2:5]`
5. `range(12,18)[-2]`
6. `len(range(25,-5,-4))`
7. `list(range(-4,-18,-3)[2:8])`
8. `list(range(3,9)[-2:-5:-1])`
9. `list(range(10,20)[-5:7])`
10. `list(range(5,20,3)[-4:4])`
11. `len(range(17,-5,-6))`
12. `18 in range(10,20,-2)`
13. `list(range(0,75,5)[2:-1:3])`
14. `list(range(10,60,4)[-2:1:-4]`
15. `range(3,12,4)[2]`
16. `len(range(2,-7,5))`

Challenge

What do each of these expressions yield?

1. `range(0,9,2)[:]`
2. `range(-2,20,5)[1:4]`
3. `range(3,18,2)[::2]`
4. `range(1,33,6)[1::2]`
5. `range(2,20)[-2:-10:-1]`
6. `range(-4,15,4)[-1:-5:-1]`
7. `range(-5,20,3)[-8:5:2]`
8. `range(100,50,-7)[1:-3:3]`

Answers

Write the expression in the Python shell and compare output.

4 | Functions

Clarify

Fill in the blanks

1.
```
#should return the sum of two numbers
def sum(x, y):
    _ x + y
```

2.
```
#should return the given string concatenated with itself
_ double_string(_):
    _ my_string + my_string
```

3.
```
#should return the sound the animal makes
_ animal_sound(_):
    if animal == 'cat':
       r_ 'meow'
    r 'unknown animal'
```

4.
```
#should return whether the number is even
_ is_even(_):
    r_ num%2 == 0
```

Exiting Early

Rewrite the following functions to use multiple `return` statements, and fewer `elif` and `else` statements.

1.
```
def factors(num):
    statement = ''
    if num%2 == 0:
  return statement = '2 is the smallest non-1 factor'
if  elif num%3 == 0:
        statement = '3 is the smallest non-1 factor'
    else:
        statement = 'neither 2 nor 3 are factors'
    return statement
```

2.
```
def contains_even(lst):
    contains = False
    for num in lst:
        if num%2 == 0:
            contains = True
    return contains
```

3.
```
def scissors_paper_rock(player1, player2):
    result = None
    if player1 == player2:
        result = 'Noone wins'
    elif (player1 == 'scissors' and player2 ==  'paper' or
            player1 == 'paper' and player2 == 'rock' or
            player1 == 'rock' and player2 == 'scissors'):
        result = 'Player1 wins'
    else:
        result = 'Player2 wins'
    return result
```

4.
```
def find_mode(x, y, z):
    mode = None
    if x==y or x==z:
        mode = x
    elif y==z:
        mode = y
    else:
        mode = (x, y, z)
    return mode
```

Print vs. Return

Identify which of the following code blocks will throw errors.

1.
```
def sum(x, y):
    print(x + y)

result = sum(2, 8)
double = result*2
```

2.
```
def letter_in_word(word, letter):
    in_word = letter in word
    return in_word

result = letter_in_word('apple', 'p')
print('Letter p in word apple: ' + str(result))
```

3.
```
def divide(x, y):
    print(x/y)

result = divide(6,3)
print('6/3 is shown above.')
```
x

4.
```
def double_string(string):
    print(string*2)

my_string = 'phone'
result = double_string(my_string)
print(my_string + ' doubled is ' + result)
```

Variable arguments

Identify which of the following code blocks will throw errors.

1. ```
   def sum(*nums):
       total = 0
       for n in nums:
           total = total + n
       return total

   sum(3, 7, 2)
   ```

2. ```
   def multiply(start_val, *nums):
       for n in nums:
           start_val = start_val * n
       return start_val

   multiply(10, 2, 2, 3)
   ```

3. ```
   def divide(numer, denom, *nums):
       total = numer/denom
       for n in nums:
           total = total / n
       return total

   divide(5)
   ```

   any number
   0 or more arguments

4. ```
   def subtract(*nums):
       total = 0
       for n in nums:
           total = total - n
       return total

   subtract()
   ```

Sequence Unpacking

Identify which of the following code blocks will throw errors.

1. ```
   def find_range(lst):
       minimum = min(lst)
       maximum = max(lst)
       return minimum, maximum

   my_min, my_max = find_range([2, 2, 7, 1, 3, 9, 3])
   ```

2. ```
   def move_point_up(x, y, z):
       return x+1, y, z

   x, y, z = move_point_up(2, 3, 1)
   ```

3. ```
   def move_point_right(x, y, z):
       return x, y+1, z

   new_point = move_point_right(2, 3, 1)
   ```

   tuple
   ```

4.
```
def move_point_down(x, y, z):
    return x-1, y, z

x, y = move_point_down(2, 3, 1)
```
3

2

5.
```
def move_point_left(x, y):
    return x, y-1

x, y, z = move_point_left(2, 3)
```

6.
```
def move_point_left(x, y):
    return x, y-1

x, y, z = move_point_left(2, 3)
```

Coding

Write a function that ...

1. ... takes integers `x` and `y` as input, where `x<y`, and returns the sum of all numbers between `x` and `y` (inclusive).
2. ... takes integers `x` and `y` as input, and returns the minimum integer (if they are the same size, than both are the minimum).
3. ... takes as input a list `lst` containing only `0`s and `1`s, and returns `True` if there are more `1`s than `0`s, else returns `False`.
4. ... takes as input a list `lst` containing only numbers, and returns the list with all numbers inside it doubled.
5. ... takes as input a list `lst` containing exactly five numbers, and returns `True` if there are three or more identical numbers in a row, else returns `False`.
6. ... takes as input a string `word`, and returns a new string identical to `word` but with all vowels removed (`a`,`e`,`i`,`o`,`u`).

Practice

Write a function that ...

1. ... takes as input three integers, `date`, `month`, `year`, and returns `True` if the three integers make a valid date, else returns `False`.
2. ... takes as input a list `numbers` containing exactly five numbers, and returns an integer that is the maximum number of identical numbers in a row.
3. ... takes as input a list `numbers` containing only numbers, and an integer `n`, and returns the sum of every n^{th} number in `numbers`.

Keep an eye on this section. It will likely grow over the semester.

Answers

Clarify

Fill in the blanks

1. ```
   #should return the sum of two numbers
   def sum(x, y):
       return x + y
   ```

2. ```
   #should return the given string concatenated with itself
   def double_string(my_string):
       return my_string + my_string
   ```

3. ```
   #should return the sound the animal makes
   def animal_sound(animal):
       if animal == 'cat':
           return 'meow'
       return 'unknown animal'
   ```

4. ```
   #should return whether the number is even
   def is_even(num):
       return num%2 == 0
   ```

Exiting Early

These are suggested alternative implementations. Note that the following implementations are not necessarily *better* than the alternatives.

1. ```
   def factors(num):
       if num%2 == 0:
           return '2 is the smallest non-1 factor'
       if num%3 == 0:
           return '3 is the smallest non-1 factor'
       return 'neither 2 nor 3 are factors'
   ```

2. ```
   def contains_even(lst):
       for num in lst:
           if num%2 == 0:
               return True
       return False
   ```

3. ```
   def scissors_paper_rock(player1, player2):
       if player1 == player2:
           return 'Noone wins'
       if (player1 == 'scissors' and player2 ==  'paper' or
               player1 == 'paper' and player2 == 'rock' or
               player1 == 'rock' and player2 == 'scissors'):
           return 'Player1 wins'
       return 'Player2 wins'
   ```
   ```

4. ```
def find_mode(x, y, z):
    if x==y or x==z:
        return x
    if y==z:
        return y
    return (x, y, z)
```

**Print vs. Return**

1. Code block throws an error, as the function call `sum(2,8)` returns `None`, and the assignment `double = None*2` throws an error.

2. Code block does not throw an error. The function call `letter_in_word('apple', 'p')` returns `True`, and the print statement `print('Letter p in word apple: '+ str(True))` succeeds in printing.

3. Code block does not throw an error. The function call `divide(6,3)` prints `2.0` and returns `None`, and the print statement `print('6/3 is shown above.')` succeeds in printing as it makes no reference to the function call.

4. Code block throws an error, as the function call `double_string('phone')` returns `None`, and the print statement `print('phone'+ 'doubled is '+ None)` throws an error.

**Variable arguments**

1. Code block does not throw an error, as the function definition `sum(*nums)` allows `*nums` to take any number of arguments, thus `sum(3, 7, 2)` is a valid function call.

2. Code block does not throw an error, as the function definition `multiply(start_val, *nums)` requires at least one argument for `start_val`, and any number of arguments for `*nums`, thus `multiply(10, 2, 2, 3)` is a valid function call.

3. Code block throws an error, as the function definition `divide(numer, denom, *nums)` requires at least two arguments, one for `numer` and one for `denom`, and any number of arguments for `*nums`, thus `divide(5)` is an invalid function call as it has insufficient arguments.

4. Code block does not throw an error, as the function definition `subtract(*nums)` allows `*nums` to take any number of arguments (including zero arguments), thus `subtract()` is a valid function call.

**Sequence Unpacking**

1. Code block does not throw an error, as the function returns two values, which are unpacked and assigned to two variables.

2. Code block does not throw an error, as the function returns three values, which are unpacked and assigned to three variables.

3. Code block does not throw an error, as the function returns three values as a tuple, and the tuple is assigned to one variable.

4. Code block throws an error, as the function returns three values, thus they cannot be unpacked into two variables.

5. Code block throws an error, as the function returns two values, thus they cannot be unpacked into three variables.

6. Code block throws an error, as the function returns two values, thus they cannot be unpacked into three variables.
```

Coding

1. ```
   def sum_range(x, y):
       return sum(range(x,y+1))
   ```

2. ```
   def minimum(x, y):
       if x<y:
           return x
       return y
   ```

3. ```
   def triples(lst):
       ones = lst.count(1)
       zeroes = lst.count(0)
       return ones > zeroes
   ```

4. ```
   def double_values(lst):
       for i in range(len(lst)):
           lst[i] = lst[i]*2
       return lst
   ```

5. ```
   def find_triple(lst):
       return (lst[0] == lst[1] and lst[1] == lst[2] or
                   lst[1] == lst[2] and lst[2] == lst[3] or
                   lst[2] == lst[3] and lst[3] == lst[4] )
   ```

6. ```
   def remove_vowels(word):
       vowels = 'aeiou'
       new_word = ''
       for char in word:
           if char not in vowels:
               new_word = new_word + char
       return new_word
   ```

Practice

Ask for assistance in tutorials and consultations.

5 | Code Traces

To be completed.

6 Recursion

Clarify

1. The following recursive function find_min returns the minimum of a list. Fill in the base case

   ```
   def find_min(lst):
       #base case
       return min(lst[0], find_min(lst[1:]))
   ```

   For example:

   ```
   >>> find_min([5, 7, 8 , -5, 10])
   -5
   ```

2. The following recursive function first_upper returns the first upper case letter of a string. It returns an empty string if it cannot find any. Fill in the base case.

   ```
   def first_upper(string):
       #base case
       else:
           return first_upper(string[1:])
   ```

   For example:

   ```
   >>> first_upper('abc Xyz)
   X
   ```

3. The following recursive function count_repeat counts how many times a target element appears in a list. Fill in the recursive case.

   ```
   def count_repeat(lst,target):
       if len(lst) == 0:
           return 0
       #recursive case
   ```

   For example:

   ```
   >>> count_repeat([15, 1, 10, 9, 10], 10)
   2
   ```

4. Write a function which returns factorial of a given number n. For example:

   ```
   >>> factorial(5)
   120
   ```

Practice

1. Write a recursive function digit_sum that sums the digits of an integer number, without converting the number to string. For example:

```
>>> digit_sum(523)
10
```

2. Write a recursive function remove_neg to remove all negative numbers from a list. For example:

```
>>> remove_neg([2, -5, 3, -7, 10])
[2, 3, 10]
```

3. Write a recursive function which returns the value of the Fibonacci series for the n^{th} element. For example:

```
>>> fib(6)
8
>>> fib(10)
55
```

Challenge

1. Write a recursive function `replace_between(string, deli, ch)` to replace all characters bounded by a delimiter deli (special word) in a string with a character ch. Note that the string contains zero or at most one special word. For example:

```
>>> replace_between('zzz$1234$xx', '$', '0')
'zzz$0000$xx'
>>> replace_between('aaa$bbb', '$', '0')
'aaa$bbb'
```

   Hints:

   – try to reduce string by one or two characters (if possible) in each recursion.
   – think about the smallest string you might have and what should be returned with such string.
   – you might have more than one base case and recursive case.

2. Write a function that returns all the permutations of a given string. For example:

```
>>> permute('abc')
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

   Hints:

   – the permutations of a single character is itself.
   – the permutations of 'abc' is given by

```
permute('abc') == ['a' + permute('bc'),
                   'b' + permute('ac'),
                   'c' + permute('ab')]
```

Answers

Clarify

1.
```
def find_min(lst):
    if len(lst) == 1:
        return lst[0]
    return min(lst[0], find_min(lst[1:]))
```

2. ```
def first_upper(string):
    if string =='':
        return ''
    elif string[0] != ' ' and string[0] == string[0].upper():
        return string[0]
    else:
        return first_upper(string[1:])
```

3. ```
def count_repeat(lst,target):
    if len(lst) == 0:
        return 0
    if lst[0] == target:
        return 1 + count_repeat(lst[1:],target)
    else:
        return count_repeat(lst[1:], target)
```

4. ```
def factorial(n):
    if n == 1 or n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

## Practice

1. ```
def digit_sum(number):
    if number == 0:
        return 0
    return number%10 + digit_sum(number//10)
```

2. ```
def remove_neg(lst):
    if len(lst) == 0:
        return []
    if lst[0]>=0:
        return [lst[0]] + remove_neg(lst[1:])
    else:
        return remove_neg(lst[1:])
```

3. ```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Challenge

1.
```
def replace_between(string, ch1, ch2):
    if len(string) <= 2:
        return string
    if string[0] == string[-1] == ch1:
        return ch1 + ch2*(len(string)-2) + ch1
    if string[0] == ch1:
        return replace_special_word(string[:-1], ch1, ch2) + string[-1]
    if string[-1] == ch1:
        return string[0] + replace_special_word(string[1:], ch1, ch2)
    return string[0] + replace_special_word(string[1:-1], ch1, ch2) + string[-1]
```

2.
```
def permute(s):
    newlst = []
    if len(s) == 1:
        return s
    else:
        for i,j in enumerate(s):
            for perm in permute(s[:i] + s[i+1:]):
                newlst += [j + perm]
    return newlst
```

Part II

Data Structures

7 | Tables and Matrices

7.1 Tables

Clarify

Code trace

What is the value of `x` after the code sequence is executed?

1.
```
table = []
for _ in range(2):
    row = [None, None]
    table.append(row)
x = table
```
0, 1

[[None, None], __]

2.
```
table = []
row = [1, 2, 3]
for _ in range(3):
    table.append(row)
table[2][2] = 5
x = table[0]
```
0, 1, 2

[1, 2, 5]

3.
```
table = []
row = [None]*3
for _ in range(4):
    table.append(row)
for i in range(len(table)):
    table[i][0] = i+1
    table[i][1] = i+2
    table[i][2] = i+3
x = table[0]
```
0, 1, 2, 3

4

[None, None, None]

1 2 3

2 3 4

3 4 5

4.
```
table = []
for _ in range(3):
    row = [1, 2]
    table.append(row)
table[1][0] = 3
x = table[0]
```
0, 1, 2

1 2

3 2

1 2

5.
```
table = []
for _ in range(3):
    row = [None]*3
    table.append(row)
for i in range(len(table)):
    table[i][0] = i+1
    table[i][1] = i+2
    table[i][2] = i+3
x = table[0]
```
0, 1, 2

3

N1 N2 N3

N N N

N N N

6.
```
table = []
for i in range(4):
    table.append([])
    for j in range(2):
        table[i].append(j)
table[2][1] = 3
x = table
```
0, 1

[[0, 1], [0, 1], [0, 1], [0, 1]]

2 3

Accessing elements

Assuming `table` is an $n \times m$ table, how can the following items be accessed?

1. The element in the first row and first column. [0] [0]
2. The (entire) second row. [1]
3. The element in the final row and first column. [n-1] [0] / [-1] [0]
4. The element in the i^{th} row and second column. [i] [1]
5. The third, fourth, and fifth element from the nineth row.

[8] [2:5]

0 1 2 3

6. The second-from-the-bottom element in the fourth column.

[-2][3]

Coding

Write a function that returns a table that ...

1. ... is $n \times n$, and all elements are `None`.
2. ... is 3×3, and each row contains the letters `'a'`, `'b'`, and `'c'` in alphabetical order.
3. ... is 6×3, and each row is a different permutation of three given items.
4. ... is 4×4, and contains the numbers 1 to 16.
5. ... is $n \times n$, and contains the numbers 1 to n^2.

_ + 4×i

0	1	2	3	4	
1	5	6	7	8	)+4
2	9	10	11	12	
3	13	14	15	16	

Practice

Code tables

Write a function that returns a table that ...

1. ... is an $n \times n$ multiplication table. E.g. if `n=3`, the function returns `[[1,2,3],[2,4,6],[3,6,9]]`.
2. ... concatenates each entry in two given sequences. E.g. if the given sequences are `'abc'` and `'12'`, the function returns `[['a1','a2'], ['b1','b2'], ['c1','c2']]`.
3. ... is a 2×2 truth table. The function should take a string version of a boolean operator as input. E.g. if `operator = 'and'`, the function returns `[[True, False],[False, False]]`.

Access and alter tables

Unless otherwise stated, each of the following functions takes as input an $n \times m$ table of numbers.

1. Implement a function that returns the mean of the row with the largest mean.
2. Implement a function that returns the mean of the column with the smallest mean.
3. Implement a function that returns `True` if at least one row contains all the same values, else returns `False`.
4. Implement a function that returns `True` if at least one column contains all the same values, else returns `False`.
5. Implement a function that takes as input an $n \times n$ table of numbers, and returns `True` if both diagonals contains all the same values, else returns `False`.
6. Implement a function that returns the mode of all values in the table.
7. Implement a function that replaces the highest and lowest value in each column with the median of that column.
8. Implement a function that replaces the highest and lowest value in each row with the mean of the row.
9. Implement a function that appends the sum of the row to the end of each row.

Challenge

Tic-Tac-Toe

Tic-tac-toe, aka noughts and crosses, is a simple two-person game where both players, X and O, take turns placing their mark, with the aim of placing three of their marks in a horizontal, vertical, or diagonal row.

Implement a function, `ttt_winning_move(board)`, that takes an $n \times n$ board, represented as an $n \times n$ table, where each row contains n items from: `'x'`, `'o'`, and `None`. The function should return a list of players with an immediate winning move, where the aim is to place n marks in a horizonal, vertical, or diagonal row. The function should return an empty list `[]` if no one has the winning move.

Knights

Chess is a two-person strategy board game where both players, black and white, take turns moving pieces, with the aim of placing the other player's king in check. One of the pieces in chess is the knight. The knight moves in an L-shape: either two squares vertically and one square horizontally, or two squares horizontally and one square vertically.

Implement a function, `move_knight(board, start, move)`, that takes an 8×8 board, represented as an 8×8 table, where each row contains 8 items from: `'K'`, and `'_'`; a tuple `start` that contains the rank and file (row and column) of the current position of the knight; and a tuple `move` that contains the rank and file where the player wishes to move the knight. If the move is legal, the function should return the board with the knight moved to its new position; else the function should return the original board.

Answers

Clarify

Code trace

1. `x == [[None, None], [None, None]]`
2. `x == [1, 2, 5]`
3. `x == [4, 5, 6]`
4. `x == [1, 2]`
5. `x == [1, 2, 3]`
6. `x == [[0,1], [0,1], [0,3], [0,1]]`

Accessing elements

1. `table[0][0]`
2. `table[1]`
3. `table[-1][0]`
4. `table[i][1]`
5. `table[8][2:5]`
6. `table[2][3]`

Coding

1.
```
def n_table(n):
    table = []
    for _ in range(n):
        table.append([None]*n)
    return table
```

2.
```
def abc_table():
    table = []
    for _ in range(3):
        table.append(['a', 'b', 'c'])
    return table
```

3.
```
def perm_table(x, y, z):
    table = []
    table.append([x,y,z])
    table.append([x,z,y])
    table.append([y,x,z])
    table.append([y,z,x])
    table.append([z,x,y])
    table.append([z,y,x])
    return table
```

4.
```
def sixteen_table():
    table = []
    for i in range(4):
        row = []
        for j in range(4):
            row = row + [i*4 + j + 1]
        table.append(row)
    return table
```

5. ```
def n_squared_table(n):
    table = []
    for i in range(n):
        row = []
        for j in range(n):
            row = row + [i*n + j + 1]
        table.append(row)
    return table
```

## Practice

Ask for assistance in tutorials and consultations.
```

7.2 Matrices

Practice

Solve the following systems of linear equations.

2x2 Linear Systems

1. **Easy Example**

$$\begin{pmatrix} 2 & 1 \\ 6 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 7 \\ 22 \end{pmatrix}$$

2. **Easy Example**

$$\begin{pmatrix} -1 & 2 \\ 4 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 7 \\ -4 \end{pmatrix}$$

3x3 Linear Systems

1. **Standard Example**

$$\begin{pmatrix} 4 & 1 & 0 \\ -4 & -2 & 2 \\ 8 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ -6 \\ 15 \end{pmatrix}$$

2. **Standard Example**

$$\begin{pmatrix} 3 & -2 & 5 \\ 9 & -2 & 16 \\ 3 & -10 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ -5 \end{pmatrix}$$

3. **Pivot Required**

$$\begin{pmatrix} 2 & -3 & 1 \\ -4 & 6 & 2 \\ 6 & -8 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -4 \\ 8 \end{pmatrix}$$

Answers

Practice

2x2 Linear Systems

1. **Easy Example**
$$\begin{pmatrix}2 & 1\\ 6 & 4\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} = \begin{pmatrix}7\\ 22\end{pmatrix}$$
Perform row operation: $R2 \leftarrow R2 - 3 \times R1$. This gives:
$$\begin{pmatrix}2 & 1\\ 0 & 1\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} = \begin{pmatrix}7\\ 1\end{pmatrix}$$
Back substitution gives $x_2 = 1$, $2x_1 + 1 = 7 \implies x_1 = 3$

2. **Easy Example**
$$\begin{pmatrix}-1 & 2\\ 4 & 4\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} = \begin{pmatrix}7\\ -4\end{pmatrix}$$
Perform row operation: $R2 \leftarrow R2 + 4 \times R1$. This gives:
$$\begin{pmatrix}-1 & 2\\ 0 & 12\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} = \begin{pmatrix}7\\ 24\end{pmatrix}$$
Back substitution gives $12x_2 = 24 \implies x_2 = 2$, $-x_1 + 2x_2 = 7 \implies x_1 = -3$

3x3 Linear Systems

1. **Standard Example**
$$\begin{pmatrix}4 & 1 & 0\\ -4 & -2 & 2\\ 8 & 1 & 3\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix} = \begin{pmatrix}6\\ -6\\ 15\end{pmatrix}$$
Perform row operations: $R2 \leftarrow R2 + R1$ and $R3 \leftarrow R3 - 2 \times R1$. This gives:
$$\begin{pmatrix}4 & 1 & 0\\ 0 & -1 & 2\\ 0 & -1 & 3\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix} = \begin{pmatrix}6\\ 0\\ 3\end{pmatrix}$$
Perform row operation: $R3 \leftarrow R3 - R2$. This gives:
$$\begin{pmatrix}4 & 1 & 0\\ 0 & -1 & 2\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix} = \begin{pmatrix}6\\ 0\\ 3\end{pmatrix}$$
Back substitution gives $x_3 = 3$, $-x_2 + 2x_3 = 0 \implies x_2 = 6$, $4x_1 + x_2 = 6 \implies x_1 = 0$.

2. **Standard Example**
$$\begin{pmatrix}3 & -2 & 5\\ 9 & -2 & 16\\ 3 & -10 & 1\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix} = \begin{pmatrix}-1\\ 0\\ -5\end{pmatrix}$$
Perform row operations: $R2 \leftarrow R2 - 3 \times R1$ and $R3 \leftarrow R3 - R1$. This gives:
$$\begin{pmatrix}3 & -2 & 5\\ 0 & 4 & 1\\ 0 & -8 & -4\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix} = \begin{pmatrix}-1\\ 3\\ -4\end{pmatrix}$$
Perform row operation: $R3 \leftarrow R3 + 2 \times R2$. This gives:
$$\begin{pmatrix}3 & -2 & 5\\ 0 & 4 & 1\\ 0 & 0 & -2\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix} = \begin{pmatrix}-1\\ 3\\ 2\end{pmatrix}$$
Back substitution gives $-2x_3 = 2 \implies x_3 = -1$, $4x_2 + x_3 = 3 \implies x_2 = 1$, $3x_1 - 2x_2 + 5x_1 = -1 \implies x_1 = 2$

3. **Pivot Required**

$$\begin{pmatrix} 2 & -3 & 1 \\ -4 & 6 & 2 \\ 6 & -8 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -4 \\ 8 \end{pmatrix}$$

Perform row operations: $R2 \leftarrow R2 + 2 \times R1$ and $R3 \leftarrow R3 - 3 \times R1$. This gives:

$$\begin{pmatrix} 2 & -3 & 1 \\ 0 & 0 & 4 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix}$$

Swap rows 2 and 3 to get:

$$\begin{pmatrix} 2 & -3 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 0 \end{pmatrix}$$

Back substitution gives $x_3 = 0$, $x_2 + x_3 = 2 \implies x_2 = 2$, $2x_1 - 3x_2 + x_3 = 2 \implies x_1 = 4$.

8 | Graphs

8.1 Paths, cycles, and connectivity

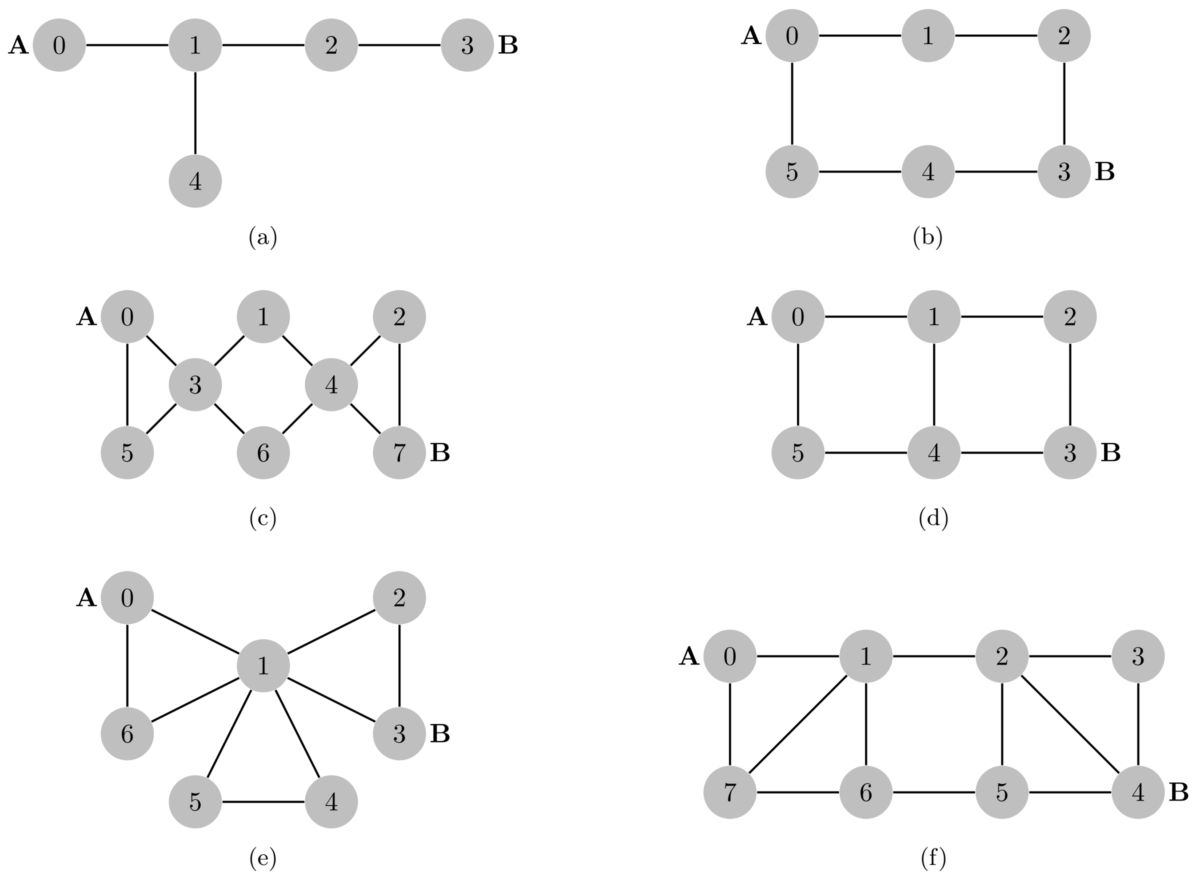
Clarify

For each of the graphs in Figure 8.1:

1. How many cycles are in the graph? Here, we consider two cycles to be identical if they contain the same vertices. That is, we do not care which vertex is the start/end vertex.

2. How many paths exist between points A and B?

3. What is the minimum number of edges which, if removed, would disconnect the graph?



(a) (b) (c) (d) (e) (f)

Figure 8.1: Graphs

Answers

Clarify

Graph 8.1a

- Cycles: 0
- Edges to remove: 1
- Paths A $\rightarrow$ B: 1
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

Graph 8.1b

- Cycles: 1
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 0$
- Edges to remove: 2
- Paths A $\rightarrow$ B: 2
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$
  - $0 \rightarrow 5 \rightarrow 4 \rightarrow 3$

Graph 8.1c

- Cycles: 3
  - $0 \rightarrow 3 \rightarrow 5 \rightarrow 0$
  - $1 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 1$
  - $2 \rightarrow 4 \rightarrow 7 \rightarrow 2$
- Edges to remove: 2
- Paths A $\rightarrow$ B: 8
  - $0 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 7$
  - $0 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 7$
  - $0 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 2 \rightarrow 7$
  - $0 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 7$
  - $0 \rightarrow 5 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 7$
  - $0 \rightarrow 5 \rightarrow 3 \rightarrow 1 \rightarrow 1 \rightarrow 7$
  - $0 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 2 \rightarrow 7$
  - $0 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 7$

Graph 8.1d

- Cycles: 3
  - $0 \rightarrow 5 \rightarrow 4 \rightarrow 1 \rightarrow 0$
  - $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 0$
- Edges to remove: 2
- Paths A $\rightarrow$ B: 4
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$
  - $0 \rightarrow 1 \rightarrow 4 \rightarrow 3$
  - $0 \rightarrow 5 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 3$
  - $0 \rightarrow 5 \rightarrow 4 \rightarrow 3$

Graph 8.1e

- Cycles: 3
  - $1 \rightarrow 0 \rightarrow 6 \rightarrow 1$
  - $1 \rightarrow 5 \rightarrow 4 \rightarrow 1$
  - $1 \rightarrow 3 \rightarrow 2 \rightarrow 1$
- Edges to remove: 2
- Paths A $\rightarrow$ B: 4
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$
  - $0 \rightarrow 1 \rightarrow 3$
  - $0 \rightarrow 6 \rightarrow 1 \rightarrow 2 \rightarrow 3$
  - $0 \rightarrow 6 \rightarrow 1 \rightarrow 3$

Graph 8.1f

- Cycles: 15
  - $0 \rightarrow 1 \rightarrow 7 \rightarrow 0$
  - $3 \rightarrow 2 \rightarrow 4 \rightarrow 3$
  - $6 \rightarrow 1 \rightarrow 7 \rightarrow 6$
  - $5 \rightarrow 2 \rightarrow 4 \rightarrow 5$
  - $0 \rightarrow 1 \rightarrow 6 \rightarrow 7 \rightarrow 0$
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0$
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0$
  - $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 1$
  - $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 1$
  - $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2$
  - $1 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 1$
  - $1 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 2 \rightarrow 1$
  - $1 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$
  - $2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 1 \rightarrow 2$
  - $2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0 \rightarrow 1 \rightarrow 2$
- Edges to remove: 2
- Paths A $\rightarrow$ B: 18
  - $0 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4$
  - $0 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 2 \rightarrow 4$
  - $0 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4$
  - $0 \rightarrow 7 \rightarrow 6 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 4$

- $0 \to 7 \to 6 \to 1 \to 2 \to 4$
- $0 \to 7 \to 6 \to 1 \to 2 \to 3 \to 4$
- $0 \to 7 \to 1 \to 6 \to 5 \to 4$
- $0 \to 7 \to 1 \to 6 \to 5 \to 2 \to 4$
- $0 \to 7 \to 1 \to 6 \to 5 \to 2 \to 3 \to 4$
- $0 \to 7 \to 1 \to 2 \to 4$
- $0 \to 7 \to 1 \to 2 \to 5 \to 4$
- $0 \to 7 \to 1 \to 2 \to 3 \to 4$
- $0 \to 1 \to 6 \to 5 \to 2 \to 3 \to 4$
- $0 \to 1 \to 6 \to 5 \to 2 \to 4$
- $0 \to 1 \to 6 \to 5 \to 4$
- $0 \to 1 \to 2 \to 3 \to 4$
- $0 \to 1 \to 2 \to 4$
- $0 \to 1 \to 2 \to 5 \to 4$

8.2 Spanning Trees

Clarify

How many spanning trees are there for the graphs in Figures 8.1b and 8.1e?

Practice

Use Prim's algorithm to find a minimum spanning tree for graph G in Figure 8.2. List the edges added to the tree after each iteration of the algorithm, and state the weight of the final spanning tree.

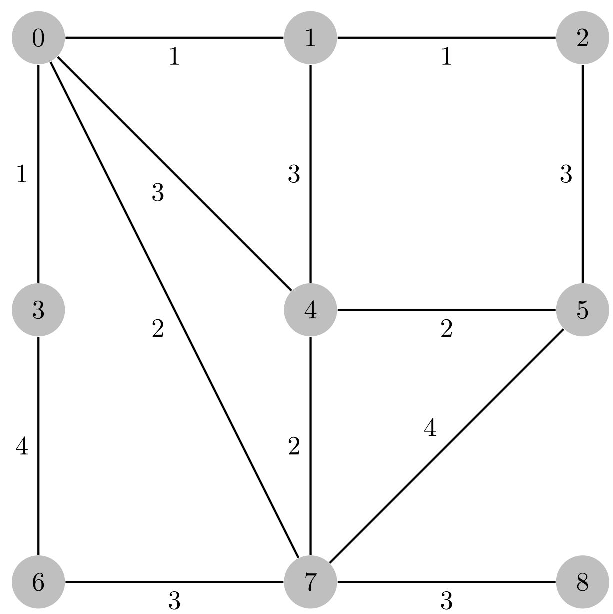


Figure 8.2: Graph G

Answers

Clarify

- The graph in Figure 8.1b has six spanning trees. Each one is a path with five edges:
  - a path from vertex 0 to vertex 5;
  - a path from vertex 1 to vertex 0;
  - a path from vertex 2 to vertex 1;
  - a path from vertex 3 to vertex 2;
  - a path from vertex 4 to vertex 3; and
  - a path from vertex 5 to vertex 4.
- The graph in Figure 8.1e has 27 spanning trees. Each triangular subgraph must be covered by exactly two edges in a spanning tree of the graph (any more will create a cycle; any fewer will result in a disconnected graph, or one that doesn't cover all the vertices). There are 3 different ways of choosing 2 edges in a triangle, and 3^3 different ways of combining these choices for all three triangles.

Practice

After each iteration, the set of edges in the partial spanning tree will be as follows (note the ordering given here is not a unique solution):

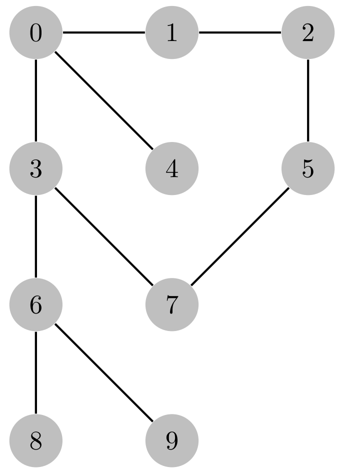
- $\{(0,1)\}$
- $\{(0,1),(0,3)\}$
- $\{(0,1),(0,3),(1,2)\}$
- $\{(0,1),(0,3),(1,2),(0,7)\}$
- $\{(0,1),(0,3),(1,2),(0,7),(4,7)\}$
- $\{(0,1),(0,3),(1,2),(0,7),(4,7),(4,5)\}$
- $\{(0,1),(0,3),(1,2),(0,7),(4,7),(4,5),(6,7)\}$
- $\{(0,1),(0,3),(1,2),(0,7),(4,7),(4,5),(6,7),(7,8)\}$

The total weight of the minimum spanning tree is 15.

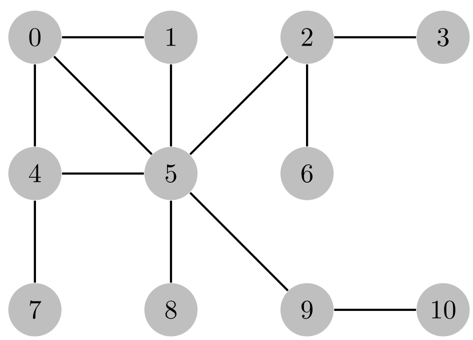
8.3 Graph Traversals

Practice

Apply Breadth First Search and Depth First Search to each of the graphs in Figure 8.3. Start your traversal from vertex 0, and write down the order in which vertices will be visited during the traversal.



(a)



(b)

Figure 8.3: Graphs for BFS and DFS

Answers

Practice

Note that these solutions are not unique — there are other possible correct orderings for both the BFS and DFS traversals.

- **Graph 8.3a**
  - BFS traversal: $0 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 8 \rightarrow 9 \rightarrow 5$
  - DFS traversal: $0 \rightarrow 3 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 7 \rightarrow 5 \rightarrow 2 \rightarrow 1 \rightarrow 4$
- **Graph 8.3b**
  - BFS traversal: $0 \rightarrow 4 \rightarrow 5 \rightarrow 1 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 2 \rightarrow 10 \rightarrow 6 \rightarrow 3$
  - DFS traversal: $0 \rightarrow 4 \rightarrow 7 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 2 \rightarrow 6 \rightarrow 3 \rightarrow 1$

9 | Stacks and Queues

to be written...

Part III

Algorithm Analysis

10 Invariants

10.1 Finding Loop Invariants

Clarify

For the following Python functions, identify the loop exit condition, and the loop invariants that, together with the loop exit condition, show that the function produces the correct output.

1.
```
def factorial(n):
    """
    Input : positive integer n
    Output: n!
    """
    res = 1
    m = 1
    while m <= n:
        res *= m
        m = m+1
    return res
```

m	res
1	1
2	1
3	1x2

$\underline{I}$: res = product$(1 \times \ldots (m-1))$

m	res
2	1
3	1x2

$\underline{Ex}$: $m = n+1$

$\underline{POC}$: res = product$(1 \times \ldots n)$

2.
```
def reverse(lst):
    """
    Input : list lst of elements
    Output: list of elements lst in reverse order
    """
    revlst = []
    while len(lst) > 0:
        revlst.append(lst.pop())
    return revlst
```

lst + reverse(revlist) = og

3.
```
def exponentiation(x,n):
    """
    Input : integers x, n
    Output: x**n
    """
    res = 1
    for i  in range(n):
        res *= x
    return res
```

0	1
0	1 x x
1	1 x x
1	1 x x x x

$\underline{I}$: x^i

$\underline{I'}$: x^{i+1}

$\underline{Ex}$: $i = n-1$

$\underline{POC}$: x^n

4.
```
def quotient_and_remainder(x,y):
    """
    Input : positive integers x, y
    Output: x//y, x%y
    """
    q=0
    r=x
    while y <= r:
        r = r-y
        q = 1+q
    return q,r
```

$x = yq + r$

$x = yq + r$

Ex: $r < y$

PoC:

$$\begin{array}{r|l} & 15 \\ \hline 3 & 47 \\ & \underline{3} \\ & 17 \\ & \underline{15} \\ & 2 \end{array}$$

$y = 3$

$r = 47$	$47-3$	
$q = 0$	1	

Answers

1. def factorial(n):

```
def factorial(n):
    """
    Input : positive integer n
    Output: n!
    """
    res = 1
    m = 1
    while m <= n:
        #Invariant: res == (m-1)!
        res *= m
        m = m+1
        #Invariant: res == (m-1)!

    #Exit condition: m == n+1
    #Post-condition: res == n!

    return res
```

2. def reverse(lst):

```
def reverse(lst):
    """
    Input : list lst of elements
    Output: list of elements lst in reverse order
    """
    revlst = []
    while len(lst) > 0:
        #Invariant: lst + reverse(revlst) gives the original list
        revlst.append(lst.pop())
        #Invariant: lst + reverse(revlst) gives the original list

    #Exit condition: lst is empty
    #Post-condition: revlst contains the entire original list in reverse order

    return revlst
```

3. def exponentiation(x,n):

```
def exponentiation(x,n):
    """
    Input : integers x, n
    Output: x**n
    """
    res = 1
    for i in range(n):
        #Invariant: x**i == res
        res *= x
        #Invariant: x**(i+1) == res

    #Exit condition: i == n - 1
    #Post-condition: res == x**n

    return res
```

4.
```
def quotient_and_remainder(x,y):
    """
    Input : positive integers x, y
    Output: x//y, x%y
    """
    q=0
    r=x
    while y <= r:
        #Invariant: x == y*q + r
        r = r-y
        q = 1+q
        #Invariant: x == y*q + r

    #Exit condition: r < y
    #Post-condition: x == y*q + r and r < y.
              In other words, q == x//y and r == x%y.

    return q,r
```

11 | Computational Complexity

11.1 Big-Oh Notation

The purpose of the big-oh notation is to succinctly describe the "order of growth" of a function. Recall the formal definition from the lecture:

Definition 1. A function $f(n)$ is in $O(g(n))$ (read "f is in big oh of g") if there are positive numbers c and n_0 such that for all $n \geq n_0$ it holds that $t(n) \leq cg(n)$.

In computational complexity analysis we apply this idea to time complexity functions $T(n)$, i.e., the functions that describe the number of elementary steps that an algorithm needs to perform for an input of size n.

Clarify

Polynomial bounds

For each of the following concrete time complexity give the tightest bound in terms of a simple polynomial function n^c using big-Oh notation. That is, determine the smallest c such that $T(n) \in \mathrm{O}(n^c)$.

1. $T(n) = n + 10$ $O(n)$
2. $T(n) = 2n$ $O(n)$
3. $T(n) = 5n + 15$ $O(n)$
4. $T(n) = n^2$ $O(n^2)$
5. $T(n) = 2n^2 + n$ $O(n^2)$
6. $T(n) = n^2 - 10n - 1$ $O(n^2)$
7. $T(n) = n/2$ $O(n)$
8. $T(n) = 10$ $O(1)$
9. $T(n) = n \mod 100$ $O(n)$

Poly-logarithmic bounds

For each of the following concrete time complexity give the tightest bound in terms of a simple poly-logarithmic function $n^c \log^d n$ using big-Oh notation. That is, determine the smallest pair c and d such that $T(n) \in \mathrm{O}\Big(n^c \log^d n\Big)$. That is, we consider $n \log^2 n$ to be smaller[1] than $n^2 \log n$. Generally, the sum log refers to the logarithm of base 2 and by $\log^2 n$ we mean $(\log n)^2$.

1. $T(n) = 10 \log n + 10$ $O(\log n)$
2. $T(n) = n + \log n$ $O(n)$
3. $T(n) = n \log n$ $O(n \log n)$
4. $T(n) = \log(10n + 100)$ $O(\log n)$
5. $T(n) = \log_{10} n$ $O(\log n)$
6. $T(n) = \log_{1.5} n$ $O(\log n)$
7. $T(n) = \log n^2$ $2 \log n = O(\log n)$
8. $T(n) = \log^2 n$ $O(\log^2 n)$
9. $T(n) = \log^2 n + n \log n$ $O(n \log n)$

$\log(10n+100) \leq n^c \log^d n$

Practice

For each of the following concrete time complexity give the tightest bound in terms of a simple poly-logarithmic function $n^c \log^d n$ using big-Oh notation (see clarify part for more details).

$O(n^2 \log n) > O(n \log^2 n) > O(n \log n) > O(\log^2 n)$
$> O(\log n)$

[1]Technically, we could say that we use the lexicographic order to determine what is a smaller pair. That is, we consider $(c, d) = (1, 2)$ to be smaller than $(c', d') = (2, 1)$.

1. $T(n) = 3,700,000$ $O(1)$
2. $T(n) = 10n^2 + 2$ $O(n^2)$
3. $T(n) = \sin(360n/(2\pi))$ $O(1)$
4. $T(n) = 10n^2 + 2\log^3 n$ $O(n^2)$
5. $T(n) = n^3/10 - n^2$ $O(n^3)$
6. $T(n) = \underline{25n} \underbrace{\cos(2\pi(n \mod 100)/100)}_{1}$ $O(n)$
7. $T(n) = \underline{n^2}/2 + 10\underline{n \log n}$ $O(n^2)$
8. $T(n) = 10n \underline{\log(2n)} + 5$ $O(n \log n)$
9. $T(n) = \underline{n}^3 + \underline{n}^2 \log(n^2)$ $O(n^3)$
10. $T(n) = 1 + \pi^2(\underline{\log} n)$ $O(\log n)$
11. $T(n) = \underline{n} + \underbrace{\sin(360n/(2\pi))}_{1} \log n$ $O(n)$
12. $T(n) = (10n^2 + 10)/n$ $O(n)$

Challenge

1. $T(n) = n^2 \log n + n \log^2 n$

   $O(n^2 \log n)$

2. $T(n) = 10\underline{\sqrt{n}} + \log n$

   $O(n^{1/2})$

Answers

Clarify

Polynomial bounds

1. $T(n) \in \mathrm{O}(n)$
2. $T(n) \in \mathrm{O}(n)$
3. $T(n) \in \mathrm{O}(n)$
4. $T(n) \in \mathrm{O}\left(n^2\right)$
5. $T(n) \in \mathrm{O}\left(n^2\right)$
6. $T(n) \in \mathrm{O}\left(n^2\right)$
7. $T(n) \in \mathrm{O}(n)$
8. $T(n) \in \mathrm{O}(1)$
9. $T(n) \in \mathrm{O}(1)$

Poly-logarithmic bounds

1. $T(n) \in \mathrm{O}(\log n)$
2. $T(n) \in \mathrm{O}(n)$
3. $T(n) \in \mathrm{O}(n \log n)$
4. $T(n) \in \mathrm{O}(\log n)$
5. $T(n) \in \mathrm{O}(\log n)$
6. $T(n) \in \mathrm{O}(\log n)$
7. $T(n) \in \mathrm{O}(\log n)$
8. $T(n) \in \mathrm{O}\left(\log^2 n\right)$
9. $T(n) \in \mathrm{O}(n \log n)$

Practice

1. $T(n) \in \mathrm{O}(1)$
2. $T(n) \in \mathrm{O}\left(n^2\right)$
3. $T(n) \in \mathrm{O}(1)$
4. $T(n) \in \mathrm{O}\left(n^2\right)$
5. $T(n) \in \mathrm{O}\left(n^3\right)$
6. $T(n) \in \mathrm{O}(n)$
7. $T(n) \in \mathrm{O}\left(n^2\right)$
8. $T(n) \in \mathrm{O}(n \log n)$
9. $T(n) \in \mathrm{O}\left(n^3\right)$
10. $T(n) \in \mathrm{O}(\log n)$
11. $T(n) \in \mathrm{O}(n)$
12. $T(n) \in \mathrm{O}(n)$

Challenge

1. $T(n) \in \mathrm{O}\left(n^2 \log n\right)$
2. $T(n) \in \mathrm{O}\left(n^{0.5}\right)$

11.2 Computational Complexity of Loops

Clarify

Determine for each variant of the function the computational complexity per line and conclude the overall computational complexity of the function in terms of the input parameter `n` (in big-oh notation) .

1. ```
   def ones(n):
       res = [1]
       while len(res) < n:   O(n)
           res += [1]   ← O(1)
       return res
   ```
   modify

2. ```
   def ones(n):
       res = [1]
       while len(res) < n:   O(n)
           res = res + [1]   ← new
       return res
   ```
   O(n²)

3. ```
   def ones(n):
       res = [1]
       while len(res) < n:   → O(log n)
           res = res + res   → 2, 4, 6, 8.... O(n)
       return res
   ```
   [1], [11]

4. ```
   def ones(n):          O(n²)
       res = [1]
       while sum(res) < n:
           res += [1]   ——— O(n)
       return res
   ```
   n

5. ```
   def ones(n):
       res = '1'
       while len(res) < n:   O(n)
           res += '1'   O(n²)
       return res
   ```

Practice

For each of the following functions identify the tightest simple bound (using big-oh notation) of the worst-case computational complexity in terms of the input size and justify your answer. Depending on the specific function this justification could be based on one or more of:

- structural properties of the worst-case input for a given input size (e.g., "in the worst case the input list is inversely ordered")
- what line(s) is/are dominating the overall computational cost and
- additional arguments as necessary.

If helpful to make your argument, you can copy+paste the code of the function into your response to annotate individual lines with comments.

0, 0, 0 | 0, 0, 1 | 0, 0, 2 ...

0, 1, 0

1. ```
   def loop_eg1(lst):
       n = len(lst)
       res = []                            O(n)
       for i in range(n):                  O(n)
           for j in range(n):              O(n)
               for k in range(n):
                   res += [lst[i]*lst[j]*lst[k]]   ← O(1)
       return res
   ```
   O(n³)

2. ```
   def loop_eg2(lst):
       n = len(lst)
       res = []
       for i in range(n):   — O(n)
           for j in range(i+1,n):   — O(n)
               res += [lst[i]*lst[j]]   — O(n²)
       return res
   ```
   ```

3.
```
def loop_eg3(lst):
    n = len(lst)
    res = []
    for i in range(0,n,2):
        for j in range(1,n,2):
            res += [lst[i]*lst[j]]
    return res
```

$O(n)$ $O(n^2)$ $O(n^2)$

4.
```
def loop_eg4(lst):
    n = len(lst)
    res = []
    for i in range(n):
        j=1
        while j<n:
            res += [lst[i]*lst[j]]
            j = j*2
    return res
```

$O(n)$ $\rightarrow O(n \log n)$

$\log_2 n \rightarrow O(\log n)$

5.
```
def loop_eg5(lst,target):
    n = len(lst)
    for i in range(n):
        for j in range(i,n):
            if lst[i]+lst[j] == target:
                return True
    return False
```

$O(n)$

0,0 1,1
0 1
⋮

$\rightarrow O(n^2)$ $O(n^2)$

Best	Worst
$O(1)$	$O(n^2)$

6.
```
def loop_eg6(lst, target):
    n = len(lst)
    res = []
    for i in range(1,n):
        if sum(lst[0:i]) > target:
            return i
    return None
```

$\rightarrow O(n)$ $O(n^2)$

$n \rightarrow$

Best	Worst
$O(1)$	$O(n^2)$

sum (2 item) $\rightarrow O(1)$

sum(n items) $\rightarrow O(n)$

Answers

Clarify

1. Complexity is $O(n)$. The loop iterates n times, and the complexity of the augmented assignment statement within the loop is only constant time, as only a single item is added to the list each time.

```
def ones(n):                  # complexity O(n)
    res = [1]                 # 1 = O(1)
    while len(res) < n:       # 1 + 1 ... + 1 (n times) = O(n)
        res += [1]            # 1 + 1 ... + 1 (n times) = O(n)
    return res                # 0 = O(0)
```

2. The loop iterates n times. Here, though, the concatenation statement inside the loop is creating a new list of length `len(res) + 1` each time, where `len(res)` ranges from $1..n-1$. So this statement is $O(n)$. Since it is executed n times, the total complexity is $O(n^2)$.

```
def ones(n):                       # complexity O(n^2)
    res = [1]                      # 1 = O(1)
    while len(res) < n:            # 1 + 1 ... + 1 (n times) = O(n)
        res = res + [1]            # 2 + 3 ... + n  = n(n+1)/2 - 1 = O(n^2)
    return res                     # 0 = O(0)
```

3. The cost of the line inside the loop is roughly $1 + 2 + 4 + \ldots + \frac{n}{2} + n$. This is less than n times the infinite geometric sum $(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots)$, so is $O(n)$.

```
def ones(n):                  # complexity O(n)
    res = [1]                 # 1 = O(1)
    while len(res) < n:       # 1 + 1 ... + 1 (log n times) = O(log n)
        res = res + res       # 2 + 4 ... + n  = O(n)
    return res                # 0 = O(0)
```

4. The loop iterates n times, and the augmented assignment statement inside the loop is only constant time. But the comparison operation occurring inside the loop condition contains a call to `sum`, which is $O(n)$. Since this comparison occurs n times, this gives an overall complexity of $O(n^2)$.

```
def ones(n):                  # complexity O(n^2)
    res = [1]                 # 1 = O(1)
    while sum(res) < n:       # 1 + 2 ... + n = n(n+1)/2 = O(n^2)
        res += [1]            # 1 + 1 ... + 1 (n times) = O(n)
    return res                # 0 = O(0)
```

5. Since there are no in-place updates for strings, as they are immutable objects in Python, the statement inside the loop requires recreating a new copy of the variable `res` each time, so its complexity is determined by the length of the string that is created, that is, $2 + 3 + 4 + \ldots + n = O(n^2)$.

```
def ones(n):                  # complexity O(n^2)
    res = '1'                 # 1 = O(1)
    while len(res) < n:       # 1 + 1 ... + 1 (n times) = O(n)
        res += '1'            # 2 + 3 ... + n  = n(n+1)/2 - 1 = O(n^2)
    return res                # 0 = O(0)
```

Practice

1. The computational complexity of the function is $O(n^3)$. This is because there are three nested loops, each of which iterates n times. The assignment operation augmenting the list is constant time, and is executed $O(n^3)$ times in the best and worst cases.

```
def loop_eg1(lst):
    n = len(lst)
    res = []
    for i in range(n):                            # O(n)
        for j in range(n):                        # O(n^2)
            for k in range(n):                    # O(n^3)
                res += [lst[i]*lst[j]*lst[k]]     # O(n^3)
    return res
```

2. The computational complexity of the function is $O(n^2)$. The outer loop iterates n times. The innermost loop iterates $(n-1)+(n-2)+\ldots 2+1$ times, which is $\frac{n(n+1)}{2}$, or $\frac{1}{2}n^2+\frac{1}{2}n$. The n^2 term still dominates the overall cost. The assignment operation inside the two nested loops is a constant time operation, and is executed $O(n^2)$ times in the best and worst cases.

```
def loop_eg2(lst):
    n = len(lst)
    res = []
    for i in range(n):                # O(n)
        for j in range(i+1,n):        # O(n^2)
            res += [lst[i]*lst[j]]    # O(n^2)
    return res
```

3. The computational complexity of the function is $O(n^2)$. The outer loop iterates $n/2$ times, and the inner loop iterates $n/2$ times for every outer iteration, which is a total of $(n/2)^2 = \frac{1}{4}n^2$ times. The assignment operation inside the two nested loops is a constant time operation, and is executed $O(n^2)$ times in the best and worst cases.

```
def loop_eg3(lst):
    n = len(lst)
    res = []
    for i in range(0,n,2):                # O(n)
        for j in range(1,n,2):            # O(n^2)
            res += [lst[i]*lst[j]]        # O(n^2)
    return res
```

4. The computational complexity of the function is $O(nlogn)$. The outer loop iterates n times. The inner loop iterates $\log n$ times for every outer iteration. The two assignments inside the nested loops are constant time, and will be executed $O(n \log n)$ times in the best and worst cases.

```
def loop_eg4(lst):
    n = len(lst)
    res = []
    for i in range(n):                # O(n)
        j=1
        while j<n:                    # O(nlogn)
            res += [lst[i]*lst[j]]    # O(nlogn)
            j = j*2                   # O(nlogn)
    return res
```

5. The computational complexity of the function is $O(n^2)$. In the worst case, no pair of elements in the list add up to the target, so the two nested loops finish iterating completely before the function returns. In this case, the outer loop iterates n times, and the innermost loop iterates $(n-1)+(n-2)+\ldots 2+1$ times, which is $\frac{n(n+1)}{2}$, or $\frac{1}{2}n^2+\frac{1}{2}n$. The assignment operation inside the two nested loops is a constant time operation which will be executed $O(n^2)$ times.

   In the best case, the first two elements in the list add up to the target, so the function returns before a single loop iteration has completed. This is $O(1)$.

```
def loop_eg5(lst,target):
    n = len(lst)
    for i in range(n):                     # O(n)
        for j in range(i,n):               # O(n^2)
            if lst[i]+lst[j] == target:    # O(n^2)
                return True
    return False
```

6. The computational complexity of the function is $O(n^2)$. In the worst case, the sum of all elements in the list is still less than the target, so the function runs through to the end before returning. In this case, the outer loop iterates n times. The call to `sum` has a complexity of $O(i)$ each time it is called, where `i` ranges from 1 to `n-1`. So this line requires $1 + 2 + 3 + \ldots + (n-2) + (n-1)$ operations in the worst case, which is $O(n^2)$.

   In the best case, the first element in the list is greater than the target, so the function returns before a single loop iteration has completed, and the call to `sum` only requires summing a single element. This is $O(1)$.

```
def loop_eg6(lst, target):
    n = len(lst)
    res = []
    for i in range(1,n):                # O(n)
        if sum(lst[0:i]) > target:      # O(n^2)
            return i
    return None
```

Part IV

Additional Resources

12 | Practice Exam Questions

The following collection of questions is an example for the middle part of the exam. It is preceded by a part where you have to write simple Python expression and apply known algorithms to specific inputs, and it is succeeded by a final part where you have to write Python programs to solve unknown problems (on the level of simple workshop problems). The recommended writing time for this middle part is 40 minutes.

12.1 Discussion of Theoretical Concepts

For each of the questions in this section, we are looking for a short answer of around 3-5 sentences. Try to be precise and coherent and use the technical terms covered in the unit where relevant.

Decrease-and-conquer

A decrease-and-conquer algorithm decreases its input in each iteration by a constant c using $O(1)$ time per iteration. That is, in the first iteration it goes from an input of size n to an input of size $n-c$. In the second iteration it goes from an input of size $n-c$ to an input of $n-2c$ and so on. Once the algorithm reaches an input of size less than c it directly solves the problem in time $O(1)$. What is the overall computational complexity of the algorithm and why?

Vertex Cover

Assume a graph $G = (V, E)$ with n vertices has a vertex cover X of size k. What can be said about the size of the largest independent set of G and why?

Complexity Classes

Recall the greedy algorithm for the Knapsack Problem that constructs solution based on some score function:

```
def greedy_knapsack(values, weights, capacity, score):
    n = len(values)
    items = sorted(range(n), key=score, reverse=True)
    sel, value, weight = [], 0, 0
    for i in items:
        if weight + weights[i] <= capacity:
            sel += [i]
            weight += weights[i]
            value += values[i]
    return sel, value, weight
```

Assume you find a score function to be used in this algorithm that

- has a computational complexity of $O(n^5)$ and
- makes the algorithm always return an optimal solution.

Based on your knowledge about the complexity classes P, NP, and NP-complete, what does this imply for the Hamiltonian Cycle Problem and why?

12.2 Computational Complexity

For each of the following functions identify the tightest simple bound (using big-oh notation) of the worst-case computational complexity in terms of the input size and justify your answer. Depending on the specific function this justification could be based on one or more of:

- structural properties of the worst-case input for a given input size (e.g., "in the worst case the input list is inversely ordered")
- what line(s) is/are dominating the overall computational cost and
- additional arguments as necessary.

If helpful to make your argument, you can copy+paste the code of the function into your response to annotate individual lines with comments.

Exponential Summary

For the following function that accepts as input a `lst`, identify the tightest simple bound of the worst-case computational complexity in terms of the input size `n = len(lst)` and justify your answer.

```
def exp_summary2(lst):
    n = len(lst)
    res = []
    i = 1
    while i <= n:        O(log n)
        res += [lst[i-1]]
        i *= 2
    return res
```

$i = 1, 2, 4, 8 \ldots$

$i-1$ $0, 1, 3, 7$

Finding Common Elements

For the following function that accepts as input two lists `lst1` and `lst2`, identify the tightest simple bound of the worst-case computational complexity in terms of the combined input size `n = len(lst1)+len(lst2)` and justify your answer.

```
def contained_in_both3(lst1, lst2):
    n1, n2 = len(lst1), len(lst2)
    lst1 = sorted(lst1)  } nlogn
    lst2 = sorted(lst2)  }            } O(n log n)
    res = []
    i, j = 0, 0
    while i < n1 and j < n2:  → O(n)
        if lst1[i] < lst2[j]:
            i += 1
        elif lst1[i] > lst2[j]:
            j += 1
        else:
            res += [lst1[i]]
            i += 1
            j += 1
    return res
```

Median Square

For the following function that accepts as input two lists `lst1` and `lst2`, identify the tightest simple bound of the worst-case computational complexity in terms of the combined input size `n = len(lst1)+len(lst2)` and justify your answer.

```
def median_square(lst1, lst2):
    squares = []
    for x in lst1:                          # O(n)
        for y in lst2:                      # O(n^2)
            squares += [x*y]                # O(n^2)

    return sorted(squares)[len(squares)//2] # O(n^2 log n)
```

$\to O(n)$ $\to O(n^2)$ $\to O(n^2 \log n)$

$\to O(n^2 \log n)$

12.3 Invariants

$n^2 \log n^2 = 2n^2 \log n = O(n \log n)$

For each of the following functions, identify the loop invariant(s), the loop exit condition, and the loop post-condition which shows the algorithm's correctness.

You can simply list the assertions, or you can copy and paste the function into the answer field and add assertion annotations as comments, as shown in the lecture.

Inversions

For the following function, identify the loop invariant(s), the loop exit condition, and the loop post-condition which shows the algorithm's correctness. You can either use the invariant at the end of the inner or the outer loop.

```
def inversions(lst):
    """
    Input : a list of comparable elements
    Output: the number of 'inversions' in lst, i.e., the number
            of index pairs i, j that violate ascending order or
            in other words: i < j and lst[i] > lst[j]

    For example:
    >>> inversions([1, 3, 2])
    1
    >>> inversions([3, 2, 1, 0])
    6
    """
    n = len(lst)
    count = 0
    for k in range(n):
        for l in range(k+1, n):
            count += lst[k] > lst[l]
    return count
```

i, j $\quad i \le k < j < l$

k L

1, 3, 2

K

3, 2, 1, 0

count = i) 1, 1

ii) 3, 2

Longest Run

For the following function, identify the loop invariant(s), the loop exit condition, and the loop post-condition which shows the algorithm's correctness. You can either use the invariant at the end of the inner or the outer loop.

```
def longest_run(lst):
    """
    Input : a list of comparable elements
    Output: a longest sublist of lst that is sorted

    For example:
    >>> longest_run([1, 2, 0, 1, 2, 1, 0, 1])
    [0, 1, 2]
    >>> longest_run([10, 9, 8, 7, 11])
    [7, 11]
    """
```

2 > 1

i 1

l j

k

1, 2, 0, 1, 2, 1, 0, 1

10, 9, 8, 7, 11

```
n = len(lst)
k, l = 0, 1
i, j = 0, 1
while j < n:
    if lst[j] < lst[j-1]:
        i = j
    j += 1
    if j - i > l - k:
        k, l = i, j
return lst[k: l]
```

lst[k:l] is the longest run in lst[:j]

j = n

lst [k:l] is the longest

13 Solutions to Practice Exam Questions

13.1 Solutions: Discussion of Theoretical Concepts

Decrease-and-conquer

Such an algorithm has an overall linear time computational complexity, because it requires n//c iterations to solve the problem. Since each iteration costs O(1), this results in a complexity of O(n//c)=O(n).

Vertex Cover

All vertices that are not in the vertex cover X form an independent set (because if there was an edge between any two of them this edge wouldn't be covered by X). There are n-k vertices that are not contained in X. Thus, the largest independent set of G must be of at least size n - k (but perhaps there are even larger independent sets).

Complexity Classes

Using this hypothetical score function would result in an overall O(n**6 log n) algorithm for the Knapsack Problem, which is polynomial time. This would also imply a polynomial time algorithm for the decision variant of the Knapsack problem, which is NP-complete. Since the Hamiltonian Cycle Problem is in NP, it can be polynomially reduced to the Knapsack Problem and, hence, we would also have a polynomial time algorithm for that problem.

13.2 Solutions: Computational Complexity

Exponential Summary

The computational complexity of the function is O(log n) irrespectively of the structure of the input list.

This is because the iteration variable is doubled in each iteration resulting in O(log n) executions of the while loop, and each individual operation inside the loop body as well as each individual check of the loop condition cost O(1).

```
def exp_summary2(lst):                # O(log n)
    n = len(lst)
    res = []
    i = 1
    while i <= n:                     # O(log n)
        res += [lst[i-1]]             # O(log n)
        i *= 2                        # O(log n)
    return res
```

Finding Common Elements

The computational worst-case complexity of the function is O(n log n), which is attained for two unsorted inputs lst1 and lst2 of roughly equal lengths n1 and n2.

In this case the two sorting steps dominate the computational cost, which are both of complexity O(n/2 log n/2) = O(n log n). In contrast, the while loop only costs O(n1+n2) = O(n) because in every iteration at least one of i or j is increased, which means that after at most n iterations either i == n1 or j == n2.

```
def contained_in_both3(lst1, lst2):  # O(n log n)
    n1, n2 = len(lst1), len(lst2)
    lst1 = sorted(lst1)              # O(n log n)
    lst2 = sorted(lst2)              # O(n log n)
    res = []
    i, j = 0, 0
    while i < n1 and j < n2:         # O(n)
        if lst1[i] < lst2[j]:        # O(n)
            i += 1                   # O(n)
        elif lst1[i] > lst2[j]:      # O(n)
            j += 1                   # O(n)
        else:
            res += [lst1[i]]         # O(n)
            i += 1                   # O(n)
            j += 1                   # O(n)
    return res
```

Median Square

The complexity of the function is O(n**2 log n) for the worst case of two input lists of roughly equal size n1=n2 (n=n1+n2).

This is because the nested for-loop create an unordered list of length n1*n2=O(n**2), which is then sorted in time O(n**2 log n**2) = O(n**2 log n) dominating the overall computational cost.

```
def median_square(lst1, lst2):
    squares = []
    for x in lst1:                             # O(n)
        for y in lst2:                         # O(n^2)
            squares += [x*y]                   # O(n^2)

    return sorted(squares)[len(squares)//2] # O(n^2 log n)
```

13.3 Solutions: Invariants

Inversions

```
def inversions(lst):
    n = len(lst)
    count = 0
    for k in range(n):
        for l in range(k+1, n):
            count += lst[k] > lst[l]
            # INV: count equal to number of index pairs i, j
            #      with i < k < j < l and lst[i] > lst[j]
    # EXC: k == l == n
    # POC: count equal to number of index pairs i, j
    #      with i < j < n and lst[i] > lst[j]
    return count
```

Longest Run

```
def longest_run(lst):
    n = len(lst)
    k, l = 0, 1
    i, j = 0, 1
```

```
while j < n:
    if lst[j] < lst[j-1]:
        i = j
    j += 1
    if j - i > l - k:
        k, l = i, j
    # INV1: lst[i:j] and lst[k: l] are sorted
    # INV2: if lst[a: b] is sorted for a<b<=j then l - k >= b - a
# EXC: j == n
# POC: if lst[a: b] is sorted for a<b<=n then l - k >= b - a
return lst[k: l]
```