CS 101: Introduction to Computer Science

Mahindra École Centrale

Duration: 1 hour 30 minutes | *Second Mid-Term Examination* | Total 100 marks

March the 25th, 2015

Problem I (*Marks:* 30)

This first problem is a list of **Multiple Choice Question**. Each question (from Q.I.1 to Q.I.15) carries 2 marks, and has one or more correct answer(s). You need to write on your answer paper the correct answers, as something like : *Problem 1: 1) a, b 2) c 3) d ... 20) a, b, d* etc on your answer paper.

TODO: 5 questions on algorithms and sorting (Q1 to Q5) and 4 questions on modules and exceptions (Q6 to Q9).

Qu.I.1) TODO: 5 questions on algorithms and sorting.

a) FIXME `print '" Yes ... " , she said. '`

b) FIXME `print " He said, « Yes! »"`

c) FIXME `print 'Hum... " That"s okay ?'"`

d) FIXME `print ' 3\'`

Qu.I.2) TODO: 5 questions on algorithms and sorting.

```
1 x = True
2 y = False
3 z = False
4 if x or (y and z):
5     print "yes"
6 else:
7     print "no"
```

a) FIXME `yes`

b) FIXME `no`

c) FIXME `SyntaxError: invalid syntax`

d) FIXME `True`

Qu.I.3) TODO: 5 questions on algorithms and sorting.

```
1 x = '5'
2 y = 2
3 z = x + y
4 print z
```

a) FIXME `SyntaxError: invalid syntax`

b) FIXME `52`

c) `7`

d) `z`

Qu.I.4) TODO: 5 questions on algorithms and sorting.

```
1 a = 6
2 b = a/2
3 print b
```

a) FIXME `SyntaxError: invalid syntax`

b) FIXME `3`

c) `6`

d) `3.0`

Qu.I.5) TODO: 5 questions on algorithms and sorting.

```
1 a = 6
2 b = a/2
3 print b
```

a) FIXME `SyntaxError: invalid syntax`

b) FIXME `3`

c) `6`

d) `3.0`

Qu.I.6) TODO: 4 questions on modules and exceptions.

a) FIXME `* this line is a comment`
b) FIXME `(that is a comment)`
c) FIXME `// that line is a comment`
d) FIXME `# that line is a comment`

Qu.I.7) TODO: 4 questions on modules and exceptions.

a) FIXME `if a >= 22:`
b) FIXME `if (a >= 22)`
c) FIXME `if (a => 22)`
d) FIXME `if a >= 22`

Qu.I.8) TODO: 4 questions on modules and exceptions.

a) FIXME `getType(example)`
b) FIXME `Type(example)`
c) FIXME `type(example)`
d) FIXME `example.type`

Qu.I.9) TODO: 4 questions on modules and exceptions.

```
1 a = input("Value of a is ")
2 if a<5:
3     blockA
4 elif a>8:
5     blockB
6 elif a>20:
7     blockC
8 else:
9     blockD
```

a) FIXME `blockA`
b) FIXME `blockB`
c) FIXME `blockC`
d) FIXME `blockD`

Qu.I.10) What is the value of this expression which uses **list comprehension**?

```
1 s = sum([ i for i in xrange(20)  if (i%2 == 0) and (i%5 == 0) ])
```

a) `s = 10`
c) `s = 30`
d) `s = [10]`
b) `s = [2, 4, 5, 6, 8, 10, 12, 15, 16, 18]`

Qu.I.11) What sequence of Python operations will produce the list `l = [2, 6, 8]` ?

a)

```
1 l = []
2 l.append(2)
3 l.append(6)
4 l.append(8)
```

b)

```
1 l = [8]
2 l.append(2)
3 l = [2] + l * 2
4 l.insert(1, 6)
5 l.pop()
```

c)

```
1 l = [8]
2 l += [6]
3 l += [2]
```

d)

```
1 l = list({ 2, 6, 8 })
2 l = l
3 l = list(l)
```

Qu.I.12) Which of the following set exist?

a) `{ -1, 1, 0, -1, 1 }`
b) `{ 2+3j, 0, 2-3j }`
c) `{ -1, [0], 1 }`
d) `{ 4, {-1, 1}, [4+6j] }`

Qu.I.13) Which of the following expression will compute the sum of positive natural numbers, strictly below 1000, that are multiples of 6 **or** 7?

a) `sum( xrange(1, 1000, 6 or 7) )`
b) `sum( xrange(1, 1000, 6) | xrange(1, 1000, 7) )`
c) `sum( set(xrange(1, 1000, 6) | set(xrange(1, 1000, 7)) )`
d) `sum( xrange(1, 1000, 6) or xrange(1, 1000, 7) )`

Qu.I.14) To create a **dictionary** that maps some integers to their cube, which of the following expression is good?

a) `d = { i**3 : i for i in xrange(12) }`
b) `d = { i : i**3 for i in xrange(12) }`
c) `d = { (i, i**3) for i in xrange(12) }`
d) `d = [ i : i**3 for i in xrange(12) ]`

Qu.I.15) If we have a (nested) **dictionary** representing a database of students (mapping roll numbers to a dictionary for each student), and you want to print all their email address, what can you try?

a)
```
1 for roll in students:
2     print "Student of roll number", roll, "has this email address:", ↩
          students['roll']['email']
```

b)
```
1 for guy in students.values():
2     print "Student of roll number", guy['roll'], "has this email address:",↩
           guy['email']
```

c)
```
1 for roll, guy in students.items():
2     print "Student of roll number", roll, "has this email address:", guy['↩
          roll']['email']
```

d)
```
1 for roll in students.values():
2     print "Student of roll number", roll, "has this email address:", ↩
          students[roll]['email']
```

Problem II (*Marks:* 20)

Problem on the recursive power : $a^n = (a \times a)^{n/2}$ if n is even.

TODO: type the problem, decide the marking scheme, write an example (to help them) ?

Qu.II.1) (*Marks:* **FIXME**)

TODO: Vipin based on the problem given by Arya.

Recursive power (it's called quick exponentiation) for numbers. Easy case when n is a power of 2: $n = 2^k$. The number of sub-calls is $\log_2(n) = k$. The number of multiplications is $O(k)$.

Remark: the semantics of your small program (the *logic* of what you write) is more important that the *syntax*.

Qu.II.2) (*Marks:* **FIXME**)

TODO: Vipin based on the problem given by Arya.

Same algorithm, but for integer power of a matrix (assuming we now how to multiply two matrices).

Remark: again, the *logic* of your program is more important that the *syntax*.

Qu.II.3) (*Marks:* **FIXME**)

TODO: Vipin based on the problem given by Arya.

General version, when n is any integer: $a^{2k} = (a \times a)^k$ and $a^{2k+1} = (a \times a)^k \times a$.

Remark: again, the *logic* of your program is more important that the *syntax*.

Qu.II.3) (*Marks:* **FIXME**)

TODO: Vipin based on the problem given by Arya.
Quick implementation of the product of two matrices (and then use it to compute `square(M)`?).
Remark: again, the *logic* of your program is more important that the *syntax*.

Problem III (*Marks:* 20)

This problem is about lists of numbers, and some mathematical operations that we can compute on such lists.

Qu.III.1) Write a simple function `max` and `min` that computes the **maximum** and **minimum** of a list `l` of numbers.

Do not use the *built-in* function `max` and `min`, **you need to program them yourself!**

You will be cautious about the case of an empty list `l = []`, for which min and max are not defined (you can raise an exception with the command `raise ValueError("min() arg is an empty sequence")`)

Qu.III.2) Similarly, write a small function `sum` that computes the **sum** of a list `l` of numbers.

Mathematically, an empty sum (ie. `sum([])`) is *defined* as 0. *Be sure to include this case.*

Again, **do not use** the built-in function `sum`, you need to program it yourself!

Qu.III.3) Now, write a function called `average` that computes (and *returns*) the arithmetic average of the numbers in the list `l`: `average`$([x_0, \dots, x_{n-1}]) = \frac{1}{n}\left(\sum_{k=0}^{n-1} x_k\right)$ for a non-empty list of size n `= len(l)`.

Be cautious about the empty list!

Examples: `average([0, 1]) = 0.5`, `average([1, 2, 3, 4, 5]) = 3.0`, and `average([-1, 1, 3]) = 1.0` for instance.

Qu.III.4) We are now interested about computing the **weighted average** of these numbers. Assume you have a list of numbers `l` $= [x_0, \dots, x_{n-1}]$, and a list of **weights** `w` $= [w_0, \dots, w_{n-1}]$, of the same sizes.

Write a function `weighted_average` that will accept two arguments `l` and `w`, and will compute (and *return*) the weighted average: `weighted_average`$([x_0, \dots, x_{n-1}], [w_0, \dots, w_{n-1}]) = \frac{\left(\sum_{k=0}^{n-1} w_k x_k\right)}{\sum_{k=0}^{n-1} w_k}$ for two non-empty lists `l, w` of the *same size* n.

You can chose to raise an exception if they have a different size (with something like `raise ValueError("weighted_average() excepts two lists of the same size.")`).

Hint: the usual average is like the weighted average with equal weights $w_0 = \dots = w_{n-1} = 1$.

Qu.III.5) What is the **time complexity** (ie. *approximate* number of elementary operations) of your `average` function?

Similarly, what is the time complexity of `weighted_average`, as a function of n, common size of the two lists `l` and `w`?

Both time complexities will be functions of n, the size of the argument list `l`. These approximated complexities should be something like $O(\log(n))$, $O(n)$, $O(n\log(n))$, $O(n^2)$ or $O(n^3)$ for example.

Qu.III.6) Now write two functions `geom_mean` and `harmo_mean` that accepts a list of **positive numbers** (ie. $x_0 > 0, \dots, x_{n-1} > 0$), and computes (and *returns*) respectively the geometrical mean and the harmonic mean.

Definitions:

- `geom_mean`$(x) = \left(\prod_{k=0}^{n-1} x_k\right)^{1/n}$ for a non-empty list `x` of size n `= len(x)`,
- `harmo_mean`$(x) = \frac{n}{\left(\sum_{k=0}^{n-1} 1/x_k\right)}$ for a non-empty list `x` of size n `= len(x)`.

Hint: No need to check that the numbers are positive or not.

Qu.III.7) (**Harder**) Similarly, write two functions for weighted geometrical mean, and weighted harmonic mean:

Definitions:

- `geom_mean_weighted`$(x, w) = \left(\prod_{k=0}^{n-1} {x_k}^{w_k}\right)^{1/\left(\sum_{k=0}^{n-1} w_k\right)}$ for two non-empty lists `x` and `w` of the same size n,
- `harmo_mean_weighted`$(x, w) = \dfrac{\sum_{k=0}^{n-1} w_k}{\left(\sum_{k=0}^{n-1} w_k / x_k\right)}$ for two non-empty lists `x` and `w` of the same size n.

Qu.III.8) Give the order of time complexity of all the function you wrote, expressed as functions of n the size of the list. This order of complexity should be something like $O(\log(n))$, $O(n)$, $O(n\log(n))$, $O(n^2)$ or $O(n^3)$ for example.

Problem IV (***Marks:*** **30)**

Blabla blabla TODO TODO
Explain the merge sort again
They need to point out mistakes
And change the program if needed
And execute it on a list of size 8 (subdividing 4 times).

- **Syntax errors** are grammatical mistakes in the language of the program – those for which the Python interpreter will throw errors. (*Note:* no errors related with indentation are present in that program.)
- Semantic errors represent mismatches between what the program is *supposed* to do, and what it is *actually* doing (Python will usually not detect such mathematical or logic errors).

Marks: One for spotting each Syntax error (there is 5), five for each Semantic error (there is 2), and 5 for correctly representing the final print statement from the program.

You need to write down in your answer sheet the locations in the programs where these errors are (line number, and how to correct these errors).

```
1  TODO: change it, remove some comments maybe
2  TODO: add some errors (both typing and semantic), like 5 typing mistakes, 2 ↩
       semantic ones
3
4  def merge(left, right):
5      """ Assumes left and right are sorted lists.
6      Puts them together in a new sorted list result which is returned.
7      """
8      result = []
9      i = 0
10     j = 0
11     # Going over a condition-based dual loop over both
12     # 'left' and 'right'
13     while i < len(left) and j < len(right):
14         if left[i] <= right[j]:
15             result.append(left[i])
16             i += 1
17         else:
18             result.append(right[j])
19             j += 1
20         # end if-else construction
21     # end while loop
22     # It is unlikely that both 'right' and 'left' have reached
23     # their max values simultaneously. Hence, below code
24     # completes the remaining values.
25     while i < len(left):
26         result.append(left[i])
27         i += 1
28     while j < len(right):
29         result.append(right[j])
30         j += 1
31     return result
32 # end of function
33
34
35 def merge_sort(s, k=-1):
36     """ Sort the list s with the merge sort (recursive) algorithm.
37     k is just used for showing the calls that are left or right."""
38     print "In merge_sort, k=", k, "for a list of size", len(s), ":", s
39     if len(s) < 2:
```

```
40          return s[:]  # new copy of the list!
41      else:
42          mid = len(s)/2  # integer division, rounded below
43          left = merge_sort(s[0:mid], k=0)  # first left half
44          right = merge_sort(s[mid:], k=1)  # second right half
45          combine = merge(left, right)
46          print "In merge_sort, I merged left and right to have a list of size"↩
                , len(combine), ":", combine
47          return combine
48      # end of if-else condition
49  # end of function
```