THE UNIVERSITY OF WESTERN AUSTRALIA

Achieve International Excellence

Computer Science and Software Engineering

SEMESTER 1, 2014 EXAMINATIONS

CITS1401
Problem Solving and Programming

FAMILY NAME: _____ GIVEN NAMES: _____

STUDENT ID: | | | | | | | | | SIGNATURE: _____

This Paper Contains: **11** pages **(including title page)**
Time allowed: **2 hours 10 minutes**

INSTRUCTIONS:
Answer all questions. The marks for the paper total 90.
Write your answers in the spaces provided on this question paper.
No other paper will be accepted for the submission of answers.
Do not write in this space.

				X	

PLEASE NOTE

Examination candidates may only bring authorised materials into the examination room. If a supervisor finds, during the examination, that you have unauthorised material, in whatever form, in the vicinity of your desk or on your person, whether in the examination room or the toilets or en route to/from the toilets, the matter will be reported to the head of school and disciplinary action will normally be taken against you. This action may result in your being deprived of any credit for this examination or even, in some cases, for the whole unit. This will apply regardless of whether the material has been used at the time it is found.

Therefore, any candidate who has brought any unauthorised material whatsoever into the examination room should declare it to the supervisor immediately. Candidates who are uncertain whether any material is authorised should ask the supervisor for clarification.

Supervisors Only – Student left at:

This page has been left intentionally blank

Q1. Arithmetic

A point on a plane can be represented as a pair of numbers *px, py* denoting its Cartesian coordinates; a rectangle whose sides are parallel to the axes can be represented as four numbers *x1, y1, x2, y2*, denoting the coordinates of its bottom-left and top-right corners. Assume that the *math* module has been imported.

(a) Define the following function. **3 marks**

def perimeter (x1, y1, x2, y2):
#perimeter returns the length of the perimeter of the rectangle x1, y1, x2, y2
e.g. *perimeter(2, –2, 10, 10) = 40.*

(b) Define the following function. **3 marks**

def crossesAxis(x1, y1, x2, y2):
#crossesAxis returns True iff the rectangle x1, y1, x2, y2 crosses either or both axes
e.g. *crossesAxis(2, –2, 10, 10) = True, crossesAxis(–10, 0, –2, 10) = False.*

(c) Define the following function. **4 marks**

def square(x1, y1, x2, y2):
#square returns a tuple of four numbers representing a square that has the
#same area as the rectangle x1, y1, x2, y2 and is centred on the same point
e.g. *square(2, –2, 18, 2) = (6, –4, 14, 4).*

```
def perimeter(x1, y1, x2, y2):
    return 2 * (x2 - x1 + y2 - y1)


def crossesAxis(x1, y1, x2, y2):
    return x1 < 0 and x2 > 0 or y1 < 0 and y2 > 0


def square(x1, y1, x2, y2):
    halfside = math.sqrt((x2 - x1) * (y2 - y1)) / 2
    cx, cy = (x1 + x2) / 2, (y1 + y2) / 2
    return (cx - halfside, cy - halfside, cx + halfside, cy + halfside)
```

1 each for area calc, divide by 2, centre calc, and building a tuple

Q2. Booleans and testing

(a) Define the following function. **4 marks**

```
def valid(x, y, z):
#valid takes three numbers and it returns True iff
#at least one of them is positive and at most one of them is negative
```

(b) Define the following function. **6 marks**

```
def test_valid():
#test_valid returns True iff valid is correct
```

test_valid should perform a set of well-chosen tests on *valid*, with at least eight tests.

```
def valid(x, y, z):
    return x > 0 and (y >= 0 or z >= 0) or \
           y > 0 and (x >= 0 or z >= 0) or \
           z > 0 and (x >= 0 or y >= 0)

def test_valid():
    s = [-1, 0, 1]
    return all ([valid(x, y, z) ==
                 (len([u for u in [x, y, z] if u > 0]) >= 1 and
                  len([u for u in [x, y, z] if u < 0]) <= 1)
                 for x in s for y in s for z in s])
```

Q3. List comprehensions

(a) Use a list comprehension to define the following function. **4 marks**

def between(x, y, z): # assume x <= y
#between returns a list holding all multiples of z between x and y inclusive
e.g. *between(6, 20, 4) = [8, 12, 16, 20]*.

(b) Use a list comprehension to define the following function. **6 marks**

def smooth(xs): # len(xs) >= 2
#smooth returns a list holding the average of each consecutive pair of elements in xs
e.g. *smooth([3, 5, 2, 6, 7, 3, 4]) = [4.0, 3.5, 4.0, 6.5, 5.0, 3.5]*.

```
def between(x, y, z):
    return [k for k in range(x, y + 1) if k % z == 0]
```

1 each for list comp, mod calc, x, y+1

```
def smooth(xs):
    return [sum(xs[k:k+2]) / 2 for k in range(len(xs) – 1)]
```

1 each for list comp and average calc; 1 for each element; 2 for range calc

Q4. List iteration

(a) Use list iteration to define the following function. **5 marks**

def ascending(xs): # assume xs isn't empty
#ascending returns a list holding the ascending elements of xs,
#i.e. the elements of xs such that all preceding elements are smaller
e.g. *ascending([2, 1, 5, 3, 5, 1, 6, 6, 7, 0]) = [2, 5, 6, 7]*.

(b) Use list iteration to define the following function. **5 marks**

def positive(xs):
#positive returns the longest prefix of xs whose sum is positive
e.g. positive([3, 2, –4, 2, –7, 5, 21]) = [3, 2, –4, 2].

```
def ascending(xs): #xs != []
    zs = [xs[0]]
    for k in xs[1:]:
        if k > zs[-1]:
            zs += [k]
    return zs
```

1 for accumulator, 1 for xs[0], 1 for xs[1:], 1 for conditional, 1 for += statement

```
def positive(xs):
    ys = []
    z = 0
    for x in xs:
        if z + x < 0:
            return ys
        else:
            z += x
            ys += [x]
    return ys
```

1 for initialisation, 1 for loop, 1 for condition, 1 for then part, 1 for else part

Q5. List iteration

A *job* that starts at time *x* and finishes at time *y* > *x* is represented by a tuple *(x, y)*. Given a list of such jobs, the *maximal activity* is determined by

- finding the job *j* that finishes earliest;
- adding *j* to the allocation;
- searching again with the list of jobs that start after *j* has finished.

Use iteration over lists to define the following function. **10 marks**

def maximal (xs):
#maximal returns a list containing the maximal activity from xs
e.g. *maximal([(6, 9), (1, 10), (2, 4), (1, 7), (5, 6), (8, 11), (9, 11)])*
= [(2, 4), (5, 6), (6, 9), (9, 11)].

Use helper functions as you feel necessary.

```
def finishesFirst(xs): #xs != []
    z = xs[0]
    for y in xs[1:]:
        if y[1] < z[1]:
            z = y
    return z
```

1 for z, 1 for loop, 1 for xs[1:], 1 for conditional

```
def maximal(xs):
    zs = []
    while xs != []:
        m = finishesFirst(xs)
        zs += [m]
        xs = [x for x in xs if x[0] >= m[1]]
    return zs
```

1 for zs, 2 for loop control, 1 for finding m, 2 for updating xs

Q6. String processing

Arithmetic allows the use of brackets to overcome the default precedence of the various operators. For an expression to be legal, the brackets must be *balanced*: every opening bracket must be paired with a closing bracket later in the expression.

A standard algorithm for checking whether a bracketed expression is legal is to count 1 for each opening bracket and –1 for each closing bracket: the expression is illegal if any time the count becomes negative, or if the count is non-zero at the end of the expression.

Define the following function. **10 marks**

def balanced(s): # s contains only "(" or ")"
#balanced returns True iff s contains a legal sequence of brackets
e.g. *balanced("") = balanced("()") = balanced("(())()") = True,*
balanced(")") = balanced("((") = balanced(")(") = False.

```
def balanced(s):
    x = 0
    for b in s:
        if b == "(":
            x += 1
        elif x > 0:
            x -= 1
        else:
            return False
    return x == 0
```

4 for statements using x, 1 for loop, 2 for conditions, 3 for return False

Q7. Dictionaries

Define the following function. **10 marks**

def partition(s): # s is all letters and spaces
#partition returns a dictionary that records all of the words from the string s and #groups them according to their length
e.g. *partition("Wales will win the World Cup twice")*
= {3: ['win', 'the', 'Cup'], 4: ['will'], 5: ['Wales', 'World', 'twice']}.

```
def partition(s):
    d = {}
    for x in s.split():
        n = len(x)
        d[n] = d.get(n, []) + [x]
    return d
```

2 for accumulator, 1 for loop, 1 for split, 1 for len, 2 for LHS, 3 for RHS

Q8. Problem-solving techniques

(a) Describe the basic principles and efficient operation of the problem-solving technique “enumeration and search”. **5 marks**

(b) Illustrate your answer to (a) with a problem that is amenable to this technique, and sketch a solution to this problem that uses the technique. **5 marks**

(a) Enumeration and search means

- Generate all possible candidate solutions;
- Check each one to see if it’s a correct solution;
- Often good to rank candidates, or to use good candidates to identify others.

(b) Decoding an encrypted message:

- Generate all possible assignments of letters;
- Check to see which one makes sense;
- Identify (un)common letters, or (un)common letter-sequences.

Q9. Problem-solving techniques

(a) Describe the basic principles and efficient operation of the problem-solving technique “divide-and-conquer”. **5 marks**

(b) Illustrate your answer to (a) with a problem that is amenable to this technique, and sketch a solution to this problem that uses the technique. **5 marks**

(a) Divide-and-conquer means

- Divide a problem instance into several smaller instances of the same problem;
- Solve each of the smaller problems separately;
- Combine these solutions to solve the original problem.

(b) Sorting a list, e.g. mergesort

- Split the list into two lists, each half as long;
- Sort the two lists separately;
- Merge the two sorted lists, by comparing the values at the head of each one.

END OF PAPER
