PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

April 2016 Examinations

CSC 148H1S
Duration — 3 hours
No aids allowed.

Student Number: |_|_|_|_|_|_|_|_|_|_|

Last Name: ____

First Name: ____

Do not *turn this page until you have received the signal to start.*
(In the meantime, please fill out the identification section above, and read the instructions below.)

This exam consists of 7 questions on 15 pages (including this one). *When you receive the signal to start, please make sure that your copy of the exam is complete.*

Please answer questions in the space provided. You will earn 20% for any question you leave blank or write "I cannot answer this question," on. You may earn substantial part marks for writing down the outline of a solution and indicating which steps are missing.

You must achieve 40% of the marks on this final exam, to pass this course.

Write your student number at the bottom of pages 2-15 of this exam.

There is a Python API at the end of this exam that you may tear off for reference.

# 1:	_____/ 8
# 2:	_____/ 6
# 3:	_____/ 8
# 4:	_____/ 8
# 5:	_____/ 8
# 6:	_____/ 8
# 7:	_____/ 8
TOTAL:	_____/54

Good Luck!

Question 1. [8 MARKS]

Implement classes TrackRunner and RunnerRoster in subparts (a) and (b). Include class and method docstrings, but no examples are required.

Part (a) [4 MARKS]

First implement class TrackRunner to record an individual runner's information. a TrackRunner must have:

- an _init_ method
- an identity string
- an age (integer)
- a record of all this runner's race results, including the distance (in metres) and time (in seconds) for each race
- a method to add a new race result
- a method to report the best (shortest) time for each distance this runner has raced

Part (b) [4 MARKS]

Implement class RunnerRoster to record information for an entire cohort of TrackRunners. A RunnerRoster must have:

- an _init_ method
- a data structure to record multiple TrackRunners
- a method to add new TrackRunners
- a method to report the fastest TrackRunner, and their time, for each distance

This page has been left intentionally (mostly) blank, in case you need space.

Question 2. [6 MARKS]

Below is a configuration of an unsolvable grid peg solitaire puzzle, where the "*" character represents pegs, and the "." character represents spaces. Allowable moves are a horizontal or vertical jump of one peg over an adjacent peg into a space, and the peg jumped over is removed.

```
* * .
* * *
* * *
```

Part (a) [3 MARKS]

Show the extensions of this configuration. Underneath each extension, show its extensions.

Part (b) [3 MARKS]

From the original configuration plus the extensions in part (a), indicate which might be the first 3 configurations considered by breadth_first_solve, and which might be the first 3 configurations considered by depth_first_solve. Briefly explain.

Question 3. [8 MARKS]

Read the API for classes **LinkedListNode** and **LinkedList**, as well as the docstring for function **merge**. Then implement function **merge**. Note that you may only use **LinkedList** and **LinkedListNode** methods that are in the API, and that **list1** and **list2** must have all attributes correctly set when you are done.

```
def merge(list1, list2):
    """
    Merge list1 and list2 by placing list2's nodes into the
    correct position in list1 to preserve ordering.  When
    complete list1 will contain all the values from both lists,
    in order, and list2 will be empty.

    Assume list1 and list2 contain comparable values in non-
    decreasing order

    @param LinkedList list1: ordered linked list
    @param LinkedList list2: ordered linked list
    @rtype: None

    >>> list1 = LinkedList()
    >>> list1.append(1)
    >>> list1.append(3)
    >>> list1.append(5)
    >>> list2 = LinkedList()
    >>> list2.append(2)
    >>> list2.append(6)
    >>> merge(list1, list2)
    >>> print(list1.front)
    1 -> 2 -> 3 -> 5 -> 6 ->|
    """
```

This page has been left intentionally (mostly) blank, in case you need space.

Question 4. [8 MARKS]

Read the docstring and example for concatenate_flat below, and then implement it.

```
def concatenate_flat(list_):
    """
    Return the concatenation, from left to right, of strings contained
    in flat (depth 1) sublists contained in list_, but no other strings.

    Assume all non-list elements of list_ or its nested sub-lists are
    strings

    @param list list_: possibly nested sub-list to concatenate from
    @rtype: str

    >>> concatenate_flat(["five", [["four", "by"], "three"], ["two"]])
    'fourbytwo'
    """
```

Question 5. [8 MARKS]

Read the declaration of class Tree in the API, and the docstring and examples of pathlength_sets below. Then implement pathlength_sets

```
def pathlength_sets(t):
    """
    Replace the value of each node in Tree t by a set containing all
    path lengths from that node to any leaf.  A path's length is the
    number of edges it contains.

    @param Tree t: tree to record path lengths in
    @rtype: None

    >>> t = Tree(5)
    >>> pathlength_sets(t)
    >>> print(t)
    {0}
    >>> t.children.append(Tree(17))
    >>> t.children.append(Tree(13, [Tree(11)]))
    >>> pathlength_sets(t)
    >>> print(t)
    {1, 2}
       {0}
       {1}
          {0}
    """
```

Question 6. [8 MARKS]

Read the API for class BinaryTree as well as the docstring and example for swap_even below. Then implement swap_even.

```
def swap_even(t, depth=0):
    """
    Swap left and right children of nodes at even depth.

    Recall that the root has depth 0, its children have depth 1,
    grandchildren have depth 2, and so on.

    @param BinaryTree t: tree to carry out swapping on.
    @param int depth: distance from the root
    @rtype: None

    >>> b1 = BinaryTree(1, BinaryTree(2, BinaryTree(3)))
    >>> b4 = BinaryTree(4, BinaryTree(5), b1)
    >>> print(b4)
        1
            2
                3
    4
        5
    <BLANKLINE>
    >>> swap_even(b4)
    >>> print(b4)
        5
    4
        1
                3
            2
    <BLANKLINE>
    """
```

Question 7. [8 MARKS]

From the list, circle the big-oh expression that gives the best upper bound for each code fragment, and briefly explain your choice.

Part (a) [2 MARKS]

```
sum, i = 0, 0
while i**2 < n:
    sum = sum + i
    i += 2
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

Part (b) [2 MARKS]

```
i, j, sum = 1, 1, 0
while i < n**3:
    while j < n:
        sum = sum + i
        j += 1
    i = i + n
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

Part (c) [2 MARKS]

```
i, sum = 0, 0
while  i < 3 * n:
    if i % 2 == 0:
        j = 1
        while j < n:
            sum = sum + j
            j = j * 2
    else:
        j = 1
        while j < n:
            sum = sum * j
            j += (n // 5)
    i = i + 5
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

Part (d) [2 MARKS]

```
i, sum = 0, 0
while i * 2 < n:
    sum = sum + i
    i = i + 1
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

This page has been left intentionally (mostly) blank, in case you need space.

This page has been left intentionally (mostly) blank, in case you need space.

Total Marks = 54

YOU MAY TEAR OFF THESE API PAGES IF YOU WISH

Short Python function/method descriptions, and classes

```
__builtins__:
  len(x) -> integer
    Return the length of the list, tuple, dict, or string x.
  max(L) -> value
    Return the largest value in L.
  min(L) -> value
    Return the smallest value in L.
  range([start], stop, [step]) -> list of integers
    Return a list containing the integers starting with start and
    ending with stop - 1 with step specifying the amount to increment
    (or decrement). If start is not specified, the list starts at 0.
    If step is not specified, the values are incremented by 1.
  sum(L) -> number
    Returns the sum of the numbers in L.

dict:
  D[k] -> value
    Return the value associated with the key k in D.
  k in d -> boolean
    Return True if k is a key in D and False otherwise.
  D.get(k) -> value
    Return D[k] if k in D, otherwise return None.
  D.keys() -> list of keys
    Return the keys of D.
  D.values() -> list of values
    Return the values associated with the keys of D.
  D.items() -> list of (key, value) pairs
    Return the (key, value) pairs of D, as 2-tuples.

float:
  float(x) -> floating point number
    Convert a string or number to a floating point number, if
    possible.

int:
  int(x) -> integer
    Convert a string or number to an integer, if possible. A floating
    point argument will be truncated towards zero.

list:
  x in L -> boolean
    Return True if x is in L and False otherwise.
  L.append(x)
    Append x to the end of list L.
  L1.extend(L2)
    Append the items in list L2 to the end of list L1.
  L.index(value) -> integer
    Return the lowest index of value in L.
  L.insert(index, x)
    Insert x at position index.
  L.pop()
    Remove and return the last item from L.
  L.remove(value)
    Remove the first occurrence of value from L.
  L.sort()
    Sort the list in ascending order.

Module random:
  randint(a, b)
    Return random integer in range [a, b], including both end points.

str:
  x in s -> boolean
    Return True if x is in s and False otherwise.
  str(x) -> string
    Convert an object into its string representation, if possible.
  S.count(sub[, start[, end]]) -> int
    Return the number of non-overlapping occurrences of substring sub
    in string S[start:end]. Optional arguments start and end are
    interpreted as in slice notation.
  S.find(sub[,i]) -> integer
    Return the lowest index in S (starting at S[i], if i is given)
    where the string sub is found or -1 if sub does not occur in S.
  S.split([sep]) -> list of strings
    Return a list of the words in S, using string sep as the separator
    and any whitespace string if sep is not specified.

set:
  {1, 2, 3, 1, 3} -> {1, 2, 3}
  s.add(...)
    Add an element to a set
  {1, 2, 3}.union({2, 4}) -> {1, 2, 3, 4}
  {1, 2, 3}.intersection({2, 4}) -> {2}
  set()
    Create a new empty set object
  x in s
    True iff x is an element of s

list comprehension:
   [<expression with x> for x in <list or other iterable>]

functional if:
   <expression 1> if <boolean condition> else <expression 2>
   -> <expression 1> if the boolean condition is True,
      otherwise <expression 2>
```

```
class LinkedListNode:
    """
    Node to be used in linked list

    === Attributes ===
    @param LinkedListNode next_: successor to this LinkedListNode
    @param object value: data this LinkedListNode represents
    """

    def __init__(self, value, next_=None):
        """
        Create LinkedListNode self with data value and successor next_.

        @param LinkedListNode self: this LinkedListNode
        @param object value: data of this linked list node
        @param LinkedListNode|None next_: successor to this LinkedListNode.
        @rtype: None
        """
        self.value, self.next_ = value, next_

    def __str__(self):
        """
        Return a user-friendly representation of this LinkedListNode.

        @param LinkedListNode self: this LinkedListNode
        @rtype: str

        >>> n = LinkedListNode(5, LinkedListNode(7))
        >>> print(n)
        5 -> 7 ->|
        """
        # start with a string s to represent current node.
        s = "{} ->".format(self.value)
        # create a reference to "walk" along the list
        current_node = self.next_
        # for each subsequent node in the list, build s
        while current_node is not None:
            s += " {} ->".format(current_node.value)
            current_node = current_node.next_
        # add "|" at the end of the list
        assert current_node is None, "unexpected non_None!!!"
        s += "|"
        return s
```

```
class LinkedList:
    """
    Collection of LinkedListNodes

    === Attributes ==
    @param: LinkedListNode front: first node of this LinkedList
    @param LinkedListNode back: last node of this LinkedList
    @param int size: number of nodes in this LinkedList
                     a non-negative integer
    """

    def __init__(self):
        """
        Create an empty linked list.

        @param LinkedList self: this LinkedList
        @rtype: None
        """
        self.front, self.back = None, None
        self.size = 0

    def append(self, value):
        """
        Insert a new LinkedListNode with value after self.back.

        @param LinkedList self: this LinkedList.
        @param object value: value of new LinkedListNode
        @rtype: None

        >>> lnk = LinkedList()
        >>> lnk.append(5)
        >>> lnk.size
        1
        >>> print(lnk.front)
        5 ->|
        >>> lnk.append(6)
        >>> lnk.size
        2
        >>> print(lnk.front)
        5 -> 6 ->|
        """
        new_node = LinkedListNode(value)
        if self.front is None:
            # append to an empty LinkedList
            self.front = self.back = new_node
        else:
            # self.back better not be None
            assert self.back, 'Unexpected None node'
            self.back.next_ = new_node
            self.back = new_node
        self.size += 1
```

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.

    === Attributes ===
    @param object value: value stored in a Tree node
    @param list[Tree] children: list of children
    """

    def __init__(self, value=None, children=None):
        """
        Create Tree self with content value and 0 or more children

        @param Tree self: this tree
        @param object value: value contained in this tree
        @param list[Tree] children: possibly-empty list of children
        @rtype: None
        """
        self.value = value
        # copy children if not None
        self.children = children.copy() if children else []


class BinaryTree:
    """
    A Binary Tree, i.e. arity 2.

    === Attributes ===
    @param object data: data in this Tree node
    @param BinaryTree|None left: left child
    @param BinaryTree|None right: right child
    """

    def __init__(self, data, left=None, right=None):
        """
        Create BinaryTree self with data and children left and right.

        None represents an empty tree.

        @param BinaryTree self: this binary tree
        @param object data: data of this node
        @param BinaryTree|None left: left child
        @param BinaryTree|None right: right child
        @rtype: None
        """
        self.data, self.left, self.right = data, left, right
```

This page has been left intentionally (mostly) blank, in case you need space.