Assignment 7

Algorithm Analysis, and Recursion – Solutions

Date Due: March 16, 2018, 10pm **Total Marks: 70**

Questions 1-4

Questions 1-4 are written questions, and you are asked to submit a single document containing the answers to all 4 questions in a file called `a7.txt`. You can submit a text file, as indicated, but PDF or RTF formats are acceptable. The rule of thumb is that you use a common file format. If the marker cannot open your file, you will get no marks.

Question 0 (5 points):

Purpose: To force the use of Version Control in Assignment 7

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 7. This is a tool that you need to be required to use, even when you don't need it, so that when you do need it, you are already familiar with it. Do the following steps.

1. Create a new PyCharm project for Assignment 7.
2. Use `Enable Version Control Integration...` to initialize Git for your project.
3. Download the Python and text files provided for you with the Assignment, and add them to your project.
4. Before you do any coding or start any other questions, make an initial commit.
5. As you work on each question, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed a question, or want to try something different. Make your most professional attempt to use the software appropriately.
6. When you are finished your assignment, open the terminal in your Assignment 6 project folder, and enter the command: `git --no-pager log` (double dash before the word 'no'). The easiest way to do this is to use PyCharm, locate PyCharm's `Terminal` panel at the bottom of the PyCharm window, and type your command-line work there.

   Note: You might have trouble with this if you are using Windows. Hopefully you are using the department's network filesystem to store your files. If so, you can log into a non-Windows computer (Linux or Mac) and do this. Just open a command-line, `cd` to your A6 folder, and run `git --no-pager log` there. If you did all your work in this folder, git will be able to see it even if you did your work on Windows. Git's information is out of sight, but in your folder.

   Note: If you are working at home on Windows, Google for how to make git available on your command-line window. You basically have to tell the command-line app where the git app is.

You may need to work in the lab for this; Git is installed there.

What to Hand In

After completing and submitting your work for Questions 5-7, open a command-line window in your Assignment 7 project folder. Run the following command in the terminal: `git --no-pager log` (double dash before the word 'no'). Git will output the full contents of your interactions with Git in the console. Copy/paste this into a text file named `a7-git.log`.

If you are working on several different computers, you may copy/paste output from all of them, and submit them as a single file. It's not the way to use git, but it is the way students work on assignments.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: The log file shows that you used Git as part of your work for Assignment 7. For full marks, your log file contains
  - Meaningful commit messages.
  - At least two commits per programming question for a total of at least 6 commits.

Solution: The submitted log might in the form of a text file, or a couple of screenshots or camera grabs of the PyCharm interface. Ideally, the log file was submitted, but there were enough problems with Windows and setting the path properly that we have to be a little generous here.

Notes to markers:

- There are two things to look for: the number of commits and the quality of the commit messages.
- The meaningful commit messages criterion: a message has to be about the work. Implemented a function, fixed a bug, started a question, added testing, those kinds of things.
- Give 4 marks for good commit messages; 2 marks for anything else.
- Students were told at least 6 commits (2 per question).
- I don't want you counting commits very carefully. Just check that they are continuing to use Git more or less purposefully.

Question 1 (5 points):

Purpose: To practice using the big-O notation.

Degree of Difficulty: Easy

Suppose you had 5 algorithms, and you analyzed each one to determine the number of steps it required, and expressed the number of steps as a function of a size parameter, as follows:

1. $f_1(n) = 1000n^2 + 3n!$
2. $f_2(n) = n^{60} + 2^n$
3. $f_3(n) = \frac{n^3}{1000} + 3078n\log(n)$
4. $f_4(n) = 6\log(n) + \frac{5n(n-1)}{2} + 10n^2\log(n)$
5. $f_5(n) = 30n^2 + 4n + n^{2.318}$

For each of the given functions, express it using big-O. For example, if $f(n) = 17n^4 + 42$ then we would write $f(n) = O(n^4)$; you could also write $f(n) \in O(n^4)$ showing that the function is in the category $O(n^4)$.

Justifications for your answers are not necessary. Just apply the rules and state the answers.

What to hand in

Include your answers in a file called `a7.txt`, though PDF and RTF files are acceptable. Clearly identify the question number and each part. If you are submitting a text file, you can write exponents such as n^2 like this: `n^2`.

Evaluation

1 mark for every correct answer. Answers that do not use the big-O notation will not be considered correct.

Solution:

1. $f_1(n) = O(n!)$
2. $f_2(n) = O(2^n)$
3. $f_3(n) = O(n^3)$
4. $f_4(n) = O(n^2\log(n))$
5. $f_5(n) = O(n^{2.318})$

Notes to markers:

- One mark each, no justification needed.
- Don't try to explain the answer. Just check for correctness.

Question 2 (4 points):

Purpose: To practice analyzing algorithms to assess their run time complexity.

Degree of Difficulty: Easy

Analyze the following pseudo-code, and answer the questions below. Each question asks you to analyze the code under the assumption of a cost for the function doSomething(). For these questions you don't need to provide a justification.

1. Consider the following loop:

```
i = 0
while i < n:
    doSomething(...)   # see below!
    i = i + 1
```

   What is the worst case time complexity (Big-O) of this loop if `doSomething()` is a function requiring $O(1)$ steps?

2. Consider the following loop:

```
i = 0
while i < n:
    doSomething(...)   # see below!
    i = i + 2
```

   What is the worst case time complexity (Big-O) of this loop if `doSomething()` is a function requiring $O(n)$ steps?

3. Consider the following loop:

```
i = 1
while i < n:
    doSomething(...)   # see below!
    i = i * 2
```

   What is the worst case time complexity (Big-O) of this loop if `doSomething()` is a function requiring $O(m)$ steps? Note: treat m and n as independent input-size parameters.

4. Consider the following loop:

```
i = n
while i > 0:
    doSomething(...)   # see below!
    i = i - 1
```

   What is the worst case time complexity (Big-O) of this loop if `doSomething()` is a function requiring $O(m!)$ steps? Note: treat m and n as independent input-size parameters.

What to hand in

Include your answer in the `a7.txt` document. Clearly mark your work using the question number. If you are submitting a text file, you can write exponents such as n^2 like this: `n^2`.

Evaluation

- 1 mark for each correct result using big-O notation. No justification needed.

Solution:

1. $O(n)$. The loop is repeated n times, and each repeat has $O(1)$ steps.
2. $O(n^2)$. The loop is repeated n times, and each repeat has $O(n)$ steps.
3. $O(m\log(n))$. The loop is repeated $\log(n)$ times, and each repeat has $O(m)$ steps.
4. $O(m!n)$ The loop is repeated n times, and each repeat has $O(m!)$ steps.

Notes to markers:

- One mark each, no justification needed.
- Don't try to explain the answer. Just check for correctness.

Question 3 (9 points):

Purpose: To practice analyzing algorithms to assess their run time complexity.

Degree of Difficulty: Easy

(a) (3 points) Analyze the following Python script, and determine the worst-case time complexity. Justify your work with a brief explanation.

```
for i in range(n):
    j = 0
    while j < n:
        print(j - i)
        j = j + 1
```

(b) (3 points) Analyze the following Python script, and determine the worst-case time complexity. Justify your work with a brief explanation.

```
for i in range(len(alist)):
    j = 0
    while j < i:
        alist[j] = alist[j] - alist[i]
        j = j + 1
```

(c) (3 points) Analyze the following Python script, and determine the worst-case time complexity. Justify your work with a brief explanation.

```
1 for i in range(len(alist)):
2     j = 1
3     while j < len(alist):
4         alist[j] = alist[j] - alist[i]
5         j = j * 2
```

Erratum: A correction was made to line 3 above. It used to be `while j < n:`

What to hand in

Include your answer in the `a7.txt` document. Clearly identify your work using the question number. If you are submitting a text file, you can write exponents such as n^2 like this: `n^2`.

Evaluation

- 1 marks for each correct result using big-O notation;
- 2 marks for each correct justification.

Solution:

1. The worst case complexity is $O(n^2)$.

   Justification: There are two loops, and the loop variables i and j are independent (they do not affect each other). Both loops run from 0 to n. The inner loop completes n iterations for each one of the outer loop's iterations. The body of the inner loop is very simple, with a subtraction, a function call, an assignment, and an addition, so lines 4-5 require $O(1)$ steps. So we have n iterations of $O(1)$ steps happening n times, so a total of $n \times n \times O(1) = O(n^2)$.

Visualization: Geometrically, this nested loop is like "tiling" a square: i controls the rows and j controls the columns, and each $O(1)$ step is a tile. The square needs n^2 tiles.

2. The worst case complexity is $O(n^2)$

   Justification. We don't know how big the `alist` is, so we'll make it an input size parameter: let n be the length of the list `alist`. The worst case complexity is $O(n^2)$. There are two loops, and the loop variables i and j are dependent (the value of i limits the range for j). The outer loop runs from 0 to $n-1$.

   The body of the inner loop is has 3 list index steps, and assignment and a subtraction, so $O(1)$. The inner loop steps j from 0 to $i-1$. In other words, the $O(1)$ of the body is repeated i times, for a total of $i \times O(1) = O(i)$ steps. The outer loop controls i, so we have the total cost:

$$\begin{aligned}\text{total} &= O(1) + O(2) + O(3) + \cdots + O(n-1) \\ &= O(1+2+3+\cdots+(n-1)) \\ &= O(n^2)\end{aligned}$$

   To do the last step above, we need the formula

$$\sum_{i=1}^{n} i = 1+2+3+\cdots+n = \frac{n(n+1)}{2}$$

   Visualization: Geometrically, this nested loop is like "tiling" a triangle: i controls the rows and j controls the columns, and each $O(1)$ step is a tile. The triangle is roughly half of the square from the previous question.

3. The worst case time complexity is $O(n \log(n))$.

   Justification. Again, we'll define an input size parameter: let n be the length of the list `alist`. The outer loop will be executed n times and the inner loop will start at 1, and double each iteration, so it will execute $\log(n)$ times. The inner loop has 7 operations per iteration, which is constant $O(1)$.

   Visualization: Geometrically, this nested loop is like "tiling" a long oddly-shaped room: i controls the rows and j controls the columns, and each $O(1)$ step is a tile. It may seem as if this oddly shaped room is a thin triangle, but it's not. The number of tiles you use for each row increases only very slowly, and it gets slower as you get farther away from the pointy end of the room. It is not remotely a triangle because only two sides are straight. The third side bends to be more and more vertical.

Notes to markers:

- This question is about recognizing loop dependence or independence, and doing the right analysis.
- A correct justification mentions both loops, and how they combine.
- It's okay to leave out the base of the logarithm, but it's also okay to write $\log_2(n)$.
- It's okay to give a very detailed count of the operations, and it's also okay to leave the detailed count out, as long as something is said about a constant number of iterations for the body of the inner loop.
- It's okay if the justification for part (b) doesn't mention the formula explicitly.

Question 4 (6 points):

Purpose: To practice analyzing algorithms to assess their run time complexity.

Degree of Difficulty: Easy

Analyze the following Python code, and determine the worst-case time complexity. Justify your work with a brief explanation.

```
1  def check_range(square):
2    """
3    Purpose:
4        Check that the square contains only the numbers 1 ... n,
5        where n is the size of the of one side of the square
6    Pre:
7        square: a 2D list of integers, n lists of n integers
8    Post: nothing
9    Return:  True if the square contains only integers 1 ... n
10            False otherwise
11   """
12   n = len(square)
13   for i in range(n):
14     for j in range(n):
15       val = square[i][j]
16       if val not in range(1, n+1):
17         return False
18   return True
```

What to hand in

Include your answer in the `a7.txt` document. Clearly identify your work using the question number.

Evaluation

- 2 marks: Your result was correct and used big-O notation.
- 1 mark: You identified a size parameter.
- 3 marks: You correctly analyzed the loop, Lines 13-17.

Solution: Here's a thorough analysis:

- The function's input is a list of lists, and since we are expecting a square, we'll use the length of one side of the square as the input size parameter. As in the program, the size will be represented by n.
- Line 12 has 2 steps, so $O(1)$. It's not at all obvious that `len()` is $O(1)$, though.
- The first outer loop (lines 13 through 17) repeats n times.
- The body of the outer loop is the whole inner loop; nothing else.
- The inner loop (lines 14 through 17) repeats n times. The counters for the inner and the outer loops are independent.

- The body of the inner loop has an if statement. This is a situation for thinking about worst case and best case.
  - The best case would be when the function returns doing the least amount of work. In other words, if the `if` condition is `True` right away. This would happen if the very first `val` were out of range. The best case would only look at the first value in the square.
  - The condition for Line 16 uses `range()` to construct a sequence whose length is n, and the relational operator `in` uses linear search to look through the constructed range to find `val`. So the worst case for line 16 is if the value is not in the range, an the number of steps is $O(n)$. This would be the best case for the function: we take the first value from the square, and look at every value in the range on line 16, and then return immediately. So the best case for the whole function is $O(n)$.
  - The worst case for the whole function would be if the function had to look at every value in the list. This would happen if the `if` condition were always `False`.
  - We're more interested in worst cases than best cases, because we don't want to be deceived by optimistic estimates. Better to know the worst that could happen and plan accordingly. That's not pessimistic; it's preparation.
  - The worst case for the function is if each value `val` is near the end of the range on line 16. Since Line 16 is a linear search, in the worst case, about $O(n)$ steps are needed for each value in the square.
- The worst case for the inner loop suggests we have to do $O(n)$ steps for every value of j. Thus the inner loop is $O(n^2)$, because of the range for j.
- The outer loop repeats the inner loop n times as well; the total cost for lines 13-17 is $n \times O(n^2) = O(n^3)$.
- The function is over on line 18. If the program got here, it completed the worst case for lines 13-17, $O(n^3)$.
- The worst-case complexity of the whole function is therefore $O(n^3)$

Notes to markers:

- A good solution will consider how the if statement on line 16 affects the whole function.
- A good solution will identify the worst case for the function, and that comes from line 16 as well.
- The best case analysis above is for explanation, but students were not required to provide the best case analysis.

Question 5 (8 points):

Purpose: To practice simple recursion on integers.

Degree of Difficulty: Easy

(a) (2 points) The Fibonacci sequence is a well-known sequence of integers that follows a pattern that can be seen to occur in many different areas of nature. The sequence looks like

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots.$$

That is, the sequence starts with 0 followed by 1, and then every number to follow is the sum of the previous two numbers. The Fibonacci numbers can be expressed as a mathematical function as follows:

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

Translate this function into Python, and test it. The function must be recursive.

Addendum: Your function should accept a non-negative n integer as input, and return the nth Fibonacci number. It should not display anything.

(b) (2 points) The Moosonacci sequence is a less well-known sequence of integers that follows a pattern that is rarely seen to occur in nature. The sequence looks like this:

$$0, 1, 2, 3, 6, 11, 20, 37, 68, 125 \ldots$$

That is, the sequence starts with 0 followed by 1, and then 2; then every number to follow is the sum of the previous *three* numbers. For example:

- $m(3) = 3 = 0 + 1 + 2$
- $m(4) = 6 = 1 + 2 + 3$
- $m(5) = 11 = 2 + 3 + 6$

Write a recursive Python function to calculate the nth number in the Moosonacci sequence. As with the Fibonacci sequence, we'll start the sequence with $m(0) = 0$.

Addendum: Your function should accept a non-negative n integer as input, and return the nth Moosonacci number. It should not display anything.

(c) (4 points) Design a recursive Python function named `substr` that takes as input a string s, a target character c, and a replacement character r, that returns a new string with every occurrence of the character t replaced by the character r. For example:

```
>>> substr('l', 'x', 'Hello, world!')
'Hexxo, worxd!'
>>> substr('o', 'i', 'Hello, world!')
'Helli, wirld!'
>>> substr('z', 'q', 'Hello, world!')
'Hello, world!'
```

If the target does not appear in the string, the returned string is identical to the original string.

Addendum: Your function should accept two single character strings and an arbitrary string as input, and return a new string with the substitutions made. It should not display anything.

Non-credit activities

You should of course test your functions before you submit. There is no credit for testing in this question, so the issue is to test enough that you are confident without wasting your time. Use the debugger and step through these functions for a few small values of n (try a base case and a non-base case). Try to identify the stack frames in the debugging window, and notice how each function call gets its own set of variables.

What to Hand In

A file called `a7q5.py` containing:

- Your recursive functions.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

This is just a warm up, and the functions are very simple.

- 2 marks: Fibonacci function. Full marks if it is recursive, zero marks otherwise.
- 2 marks: Moosonacci function. Full marks if it is recursive, zero marks otherwise.
- 4 marks: `subst` function. Full marks if it is recursive, and if it works, zero marks otherwise.

Solution:

Fibonacci
The function can be found all over the internet, and in practically any text book used to teach recursion. The only reason it's here is because it gives students a chance to make mistakes that can easily be found and fixed.

```
def fibonacci(n):
    """
    Purpose:
        The Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, ...
    Preconditions:
        :param n: a non-negative integer
    Return:
        :return: the nth Fibonacci number, starting with fib(0) = 0
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Notes for markers:

- Full marks unless you see an error in the code.
- There was no requirement for testing.

Moosonacci
This is made up for assignments, and while it may or may not be used by other courses, the name used here is absolutely idiosyncratic.

```
def moosonacci(n):
    """
    Purpose:
        Calculate the nth Moosonacci number
        The Moosonacci numbers are: 0, 1, 2, 3, 6, 11, 20, 37, ...
    Preconditions:
        :param n: a non-negative integer
    Return:
        :return: the nth Moosonacci number, starting with moos(0) = 0
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return moosonacci(n-1) + moosonacci(n-2) + moosonacci(n-3)
```

Notes for markers:

- Full marks unless you see an error in the code.
- There was no requirement for testing.

substr
Here's a model solution:

```
def substr(t, r, s):
    """
    Purpose:
        Return a string that has the same characters in s, except that
        every occurrence of character t is replaced by character r.
    Preconditions
        :param t: the target character to replace
        :param r: the replacement character
        :param s: a string
    Return
        :return: a new string with t replaced by r in s
    """
    if len(s)==0:
        return ''
    elif s[0] == t:
        return r + substr(t, r, s[1:])
    else:
        return s[0] + substr(t, r, s[1:])
```

Notes for markers:

- Full marks unless you see an error in the code.
- There was no requirement for testing.

Question 6 (18 points):

Purpose: Practical recursion using the Node ADT.

Degree of Difficulty: Moderate

We'll be three recursive functions that work on node-chains (**not Linked Lists**), described below. You must implement them using the node ADT, and you must use recursion. Keep in mind that the node ADT is recursively designed, since the `next` field refers to another node-chain (possibly empty).

You will implement the following 3 functions:

(a) (6 points) `subst(chain, t, r)`: Modify the given node-chain, so that every occurrence of data value `t` (the target) is replaced with the data value `r` (the replacement).

Addendum: Your function should modify the node chain, and return `None`.

To test this function, create the following test cases:

- An empty chain.
- A chain with one node. Two tests here:
  1. The target is a value in the chain.
  2. The target is a value not in the chain.
- A chain with several nodes. Five tests here:
  1. The target is the first value in the chain.
  2. The target is the last value in the chain.
  3. The target is a value not in the chain.
  4. The target is a value that appears once in the chain.
  5. The target is a value that appears several times in the chain.

(b) (6 points) `reverse(chain)`: The order of the data values in the node-chain is reversed.

Addendum: Your function should reverse the node chain, and return the reference to the first node in the chain.

To test this function, create the following test cases:

- An empty chain.
- A chain with one node.
- A chain with several nodes.

(c) (6 points) `copy(chain)`: A new node-chain is created, with the same values, in the same order, but it's a separate distinct chain. Adding or removing something from the copy must not affect the original chain.

Addendum: Your function should copy the node chain, and return the reference to the first node in the new chain.

To test this function, create the following test cases:

- An empty chain.
- A chain with one node.
- A chain with one several nodes.

Be sure to check that you have two distinct chains with the same values!

What to Hand In

A file called `a7q6.py` containing:

- Your recursive functions for `subst(chain, t, r)`, `reverse(chain)`, `copy(chain)`.
- A test-script, including the cases above, and any other tests you consider important.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- (3 marks each) Your `subst(chain, t, r)`, `reverse(chain)`, `copy(chain)` functions are recursive, and correctly implement the intended behaviour.
- (3 marks each) You implemented the test cases given above.

Solution:

subst

The first function, subst, is the easiest. The function just has to step through the node chain. The size of the chain doesn't change, and the first and last nodes do not change.

```
def subst(anchor, t, r):
    """
    Purpose
        Replace every occurrence of t in the chain with the value r
    Preconditions:
        :param anchor: a node-chain
        :param t: the target value to be replaced
        :param r: the replacement value
    Post-conditions:
        Every occurrence of t is replaced by r
    Return:
        :return: None
    """

    if anchor is None:
        return
    else:
        data = node.get_data(anchor)
        if data == t:
            node.set_data(anchor,r)
        subst(node.get_next(anchor), t, r)
```

Any recursive variation is allowable.

reverse

A good solution to reverse follows. But it's not the easiest to come up with. Which is why it's a good solution. It simply pops nodes from the front of one chain and pushes them on the front of another. This is very like a stack, without needing the Stack ADT.

```
def reverse_b(to_reverse, reversed=None):
    """
    Purpose
        Move all of the nodes in to_reverse to reversed
    Preconditions:
        :param to_reverse: a node chain to be reversed
        :param reversed: a node chain already reversed
    Post-conditions:
        to_reverse is reversed
    Return:
        :return: the resulting node chain with all nodes
    """

    # "pop" off a node from q, push it on s
    if to_reverse == None:
        return reversed
    else:
     # conceptually, grab the first node in to_reverse
        this_node = to_reverse
        # conceptually, grab the nodes after the first
        the_rest = node.get_next(this_node)
        # connect the first node to the ones that are already reversed
        node.set_next(this_node, reversed)
        # recursively reverse all the rest
        return reverse_b(the_rest, this_node)
```

Notice the use of defaults in a keyword parameter! This could also be done with an internal definition.

There is a very interesting solution that takes advantage of references.

```
def reverse_a(chain):
    """
    Purpose
        Reverse the sequence of the nodes in the given chain.
    Preconditions:
        :param chain: a node chain
    Post-conditions:
        The values in the chain are in the reversed order.
    Return:
        :return: the reversed node chain
    """

    if chain is None:
        return None
    elif node.get_next(chain) is None:
        return chain
    else:
        # remember the first two nodes
        first = chain
        second = node.get_next(chain)
        # reverse the chain starting at the second node
        reved = reverse_a(second)
        # second is now last!  hook the first node
        node.set_next(second, first)
        node.set_next(first, None)
        return reved
```

WHen the shorter list is reversed, the variable `second` now refers to the last node in the reversed list. It's a simple matter to add the remaining node to the end of it. Simple to do, if you understand the effect of reverse, but not so simple to see if you're not drawing lots of diagrams.

It is going to be far more common for student submissions to include a loop to walk down a node chain, and attach the first node to the end of the reversed chain. This is completely acceptable as practice thinking about recursion. However, this version with a loop is more expensive: in total $O(n^2)$, whereas the one above is $O(n)$.

The reverse function reuses the nodes, and simply changes the references stored. As a result, we return the reversed chain, because the node that used to be at the front is now at the end. Any variable that refers to this node will now refer to the end of the reversed chain.

The copy function needed to create new nodes. The copy function should not change the original.

```
def copy(chain):
    """
    Purpose
```

```
        Make a completely new copy of the given node-chain.
    Preconditions:
        :param chain: a node-chain
    Post-conditions:
        None
    Return:
        :return: A new copy of the chain is returned
    """
    if chain == None:
        return None
    else:
        newnode = node.create(node.get_data(chain))
        newnext = copy(node.get_next(chain))
        node.set_next(newnode, newnext)
        return newnode
```

There's nothing exciting about this. I should have made it first in the sequence of exercises for the question. **Notes for markers:**

- Check if the function looks plausible. Full marks unless you see an error in the code.
- If recursion is not used, deduct 3 marks per function where it is not used.
- Testing was given as a requirement, so check that all the required tests were done.

Question 7 (15 points):

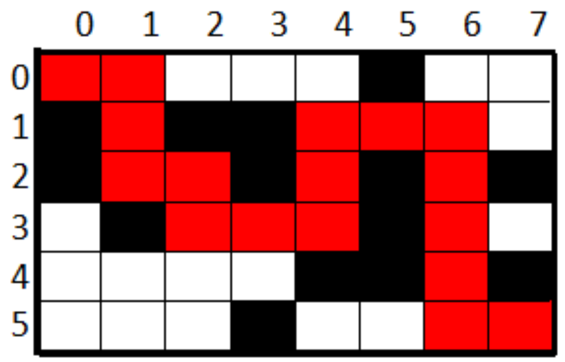
Purpose:

Degree of Difficulty: Moderate to Tricky

Problem:

A maze is a rectangular arrangement of blocks (walls) and open spaces (paths). A maze may have a path through it from a given start location to a given end location. You can usually move only one cell at a time. From each cell you choose your next move. Moves can only be into unblocked cells. In a restricted maze (which we are describing here) you can not cross your existing path: i.e., once you visit a particular cell, you can?t visit it again (for all intents and purposes, a visited cell is blocked to future attempts to move).

For example, this maze:



Has a path: `(0,0),(0,1)(1,1)(2,1)(2,2)(3,2)(3,3)(3,4)(2,4)(1,4)(1,5)(1,6)(2,6)(3,6)(4,6)(5,6)(5,7)` from start=`(0,0)` to goal=(5,7).

Solution:

Implement a Python function `MazeSolver(m,s,g)` to determine if a path exists within the maze, `m`, from the start location `s` to the end location `g`. The function must be RECURSIVE. Some constraints on your solution:

- Your maze is to be represented as a Python list of lists (recall the magic square and sudoku requirements from an earlier assignment).
- Start and goal locations are to be represented as Python tuples `(x,y)` where `x` and `y` represent the index values into the list of the cell location.
- Input to your application will be a file of text containing `'0'` for an open cell and `'1'` for a blocked cell. For example the above maze will be in a file as:

```
0 0 0 0 0 1 0 0
1 0 1 1 0 0 0 0
1 0 0 1 0 1 0 1
0 1 0 0 0 1 0 0
0 0 0 0 1 1 0 1
0 0 1 0 0 0 0 0
```

- Your output must include:
  - The maze with the path showing if a path exists (mark the path with `'P'`)

- Your function returns the boolean value `True` if a path exists, `False` otherwise

The recursive definition of the `MazeSolver()` is given by:

$$MazeSolver(m_0, s_0, g) = \begin{cases} T, output\ maze & if\ current\ location\ is\ available\ and = g \\ F & if\ current\ location\ is\ unavailable\ or\ unreachable \\ MazeSolver(m_i, s_i, g) & where\ m_i\ is\ the\ maze\ with\ current\ cell\ blocked\ and\ s_i\ is\ a\ reachable\ cell\ from\ s_0 \end{cases}$$

(NOTE: The recursive nature of the solution is that each time the path consumes a location, the new maze is a smaller version of the original, i.e. it contains fewer open cells to try.)

Reachable cells are defined as any cell in one of the four cardinal directions (north, west, south, east), except where:

- You can not go west of the left most cell of any row,
- You can not go east of the right most cell of any row,
- You can not go north of the top most cell of any row,
- You can not go south of the bottom most cell of any row,
- You can not occupy a cell already occupied by your current path.

Examples

You are given a series of mazes to solve.

- `maze1.txt` Start (0,3) Goal (4,5)
- `maze2.txt` Start (0,0) Goal (8,9)
- `maze3.txt` Start (3,0) Goal (23,30)

The three mazes provided may or may not have a path from start to goal. You can also create your own for testing purposes.

What to Hand In

- Your code for `MazeSolver()` as `MazeSolver.py`
- Screen capture shots of the output generated by your code when the maze has a path from start to finish.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 3 marks: Your script correctly reads in the maze data from a text file, and produces a list of lists using the data.
- 3 marks: You're using tuples to represent position, including start and goal.
- 9 marks: Your mazeSolver function is recursive, and correctly determines and displays a path from start to goal, if one exists.

Extra work:

Modify your code so that it will find ALL paths that meet the requirements.

Solution: A variety of solutions are found in the Solutions folder. Note the elegance of the recursive solutions. Code that is significantly longer or less well-designed will not receive full marks.

Notes for markers:

- Check that the file is being read. Lists of lists are necessary; numpy arrays will not get full marks.
- Check that tuples are being used for locations. Some students will be using tuple assignment, but others will be using indexing. Indexing is uglier, but still allowable.
- For the actual MazeSolver code, don't spend a lot of time trying to figure out the code. I have seen what some students wrote, and it's not going to be pleasant to grade. Give full marks for something that looks like the model solutions, and give them generously.
- Deduct marks if you can tell the code is not going to work, or if the code is much longer than the model solutions, or much too arcane for good design.
- It will be better to be generous than to be strict here. Maybe if you are being generous, you might add a comment like "Being generous for effort, but the code is a mess. See the model solutions!"