UNIVERSITY OF SASKATCHEWAN
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 7

Algorithm Analysis, and Recursion

Date Due: July 12, 11pm **Total Marks: 73**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M.
- Make sure your name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- Do not submit folders, or zip files, even if you think it will help.
- Programs must be written in Python 3.
- **Assignments must be submitted to Moodle.** There is a link on the course webpage that shows you how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Read the purpose of each question. Read the Evaluation section of each question.

Questions 1-4, Question 7 parts (c) and (d)

Questions 1-4, Question 7 parts (c) and (d) are written questions, and you are asked to submit a single document containing the answers to all 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.

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. $f_1(n) = n^{49} + 200^n$
2. $f_2(n) = \frac{n^8}{362} + 239n\log(n)$
3. $f_3(n) = 70500n^2 + 105n!$
4. $f_4(n) = 3\log(n) + \frac{2n^2(n-1)}{2} + 40n^3\log(n)$
5. $f_5(n) = 8n^4 + n^{4.123} + 16n$

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.

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.

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
```

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.

Question 4 (6 points):

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

Degree of Difficulty: Easy

Phoenix Wright, ace attorney, has just setup a new computer filing system to keep track of his court cases. He had his friend Larry Butz write the code (below) for the filing system. Larry claims his code is super fast, and runs in O(1) time. Is this statement at all correct? What about in the worst CASE? Best CASE? (no need for average case). JUSTICE-ify your answer. Carefully read the evaluation criteria of the question.

```
1  def file_case(case, closed_cases):
2      """
3      Purpose:
4          Checks whether the given case has been closed,
5          and if so puts it into the closed_cases.
6          If the case has not yet been closed, a search
7          is done to return any previously closed and related cases.
8      Pre:
9          case: a dictionary containing 3 fields -
10             status = a string. Either "Open" or "Closed"
11             keywords = list of strings. May match tags of other cases.
12                 Can be any arbitrary length.
13             crime = string. What the single alleged crime is.
14                 E.g. "murder", "burglary", etc.
15         closed_cases: list of cases (see case above)
16     Post: case is appended to closed_cases is if the status
17         of case is "Closed"
18     Return:  Empty list if case's status was "Closed",
19         otherwise a list of related cases
20     """
21     if case["status"] == "Closed":
22         # Case is closed. Just add it to our list of closed cases
23         closed_cases.append(case)
24         return []
25     # Case is not closed, do a search to return any related closed cases
26     related_cases = []
27     for search_term in case["keywords"]:
28         for closed_case in closed_cases:
29             if search_term == closed_case["crime"]:
30                 related_cases.append(closed_case)
31     return related_cases
```

What to hand in

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

Evaluation

- 2 marks: You correctly identified the best and worst case in Big-O, and whether Larry sucks.
- 2 mark: You identified any size input size parameters.
- 2 marks: Your JUSTICE-ifications and analysis are correct (accusing people and OBJECTIONS! are encouraged).

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.
Your function should only accept a non-negative n integer as input, and return the nth Fibonacci number. No other parameters are allowed. 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 $m(0) = 0$.
Your function should only accept a non-negative n integer as input, and return the nth Moosonacci number. No other parameters are allowed. 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.

Question 6 (18 points):

Purpose: Practical recursion using the Node ADT.

Degree of Difficulty: Moderate

Below are three recursive functions that work on node-chains (**not Linked Lists**). You MUST implement them using the node ADT, and you MUST use recursion. We will impose very strict rules on implementing these functions which will hopefully show you new ways to think about recursion. Keep in mind that the node ADT is recursively designed, since the `next` field refers to another node-chain (possibly empty).

For all of the questions you are **not** allowed to use any data collections (lists, stacks, queues). Instead, recursively pass any needed information using the PARAMETERS. DO NOT ADD ANY EXTRA PARAMETERS. None are needed.

You will implement the following three functions:

(a) (6 points) `to_string(node_chain)`: For this function, you are going to re-implement the `to_string()` operation from Assignment 5 using recursion. Recall, the function does not do any console output. It should return a string that represents the node-chain (e.g. `[ 1 | * ]-->[ 2 | * ]-->[ 3 | / ]`). Additionally, for a completely empty chain, the `to_string()` should return the string `EMPTY`.

To test this function, create the following test cases:

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

(b) (6 points) `copy(node_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. 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!

(c) (6 points) `replace(node_chain, target, replacement)`: Replace every occurrence of the data `target` in `node_chain` with `replacement`. Your function should 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 no replacements
- A chain with several replacements.

What to Hand In

A file called `a7q6.py` containing:

- Your recursive functions for `to_string()`, `copy()`, `replace()` in a file called `a7q6.py`
- A test-script called `a7q6_testing.py`, 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 `to_string()`, `copy()`, `replace()` functions are recursive, and correctly implement the intended behaviour.
- (3 marks each) You implemented the test cases given above.

Question 7 (18 points):

Purpose: To emphasize that recursion is not about code, or functions or Python. It's about relationships.

Degree of Difficulty: Tricky

Problem:

This question is designed to emphasize and reward the use of relationships in designing recursive functions. It's straight-forward if you look at it from a certain point of view, but insidiously difficult if you insist that staring at a computer is a good way to solve problems. The function that answers this problem can be written in 5-8 lines of Python code (not counting the doc-string interface). The entire problem is figuring out what the relationships are. And the code is absolutely trivial once you have the relationship right. You will know you have the relationships right because they make sense as relationships, not as Python code.

A small part of your grade for this question is the actual code, but much more weight is given to whether or not you can explain the relationship that the function makes use of.

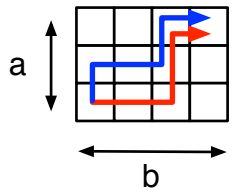
The problem starts with a room. The room will always be rectangular (a square is a special kind of rectangle, so they're allowed), but we will vary the dimensions. The room will be subdivided into an integer number of squares of equal size. Think of the squares as "tiles." In general, the room will be a tiles wide, and b tiles long. Furthermore, a and b are integers, and always positive ($a > 0$ and $b > 0$).

Mario is in the room, trying to get from one corner of the room to the other. He will always start on the tile in the lower left corner of the room, and he will always have to get to the tile in the top right corner of the room. Mario's movement is constrained by the rule that he cannot move diagonally from one tile to another. Mario can only move to an adjacent tile on the same row, or on the same column, but not diagonally. And of course, Mario cannot step outside the room.

Mario wants to follow a path that is as short as possible, counting tiles as distance. A path is a sequence of tiles that Mario can move to. The shortest path has $a + b - 1$ tiles on it, including the starting tile and the ending tile. Furthermore, and this is where the fun begins, there are possibly several paths that are all equally short, with the same number of tiles.

These paths all share the same property: on a shortest path, Mario can take a step to an adjacent tile, and he can only move up one tile (up, or "north"), or to the right one tile (right, or "east"). If Mario moves ("south"), or left ("west"), he cannot be on a shortest path.

In the diagram below, we show a 3×4 room, and two valid shortest paths that Mario could take.



The question is, in a room of $a \times b$ tiles, how many different shortest paths could he take (obeying his movement constraints, above)? We want to count paths that are different, and we want to count all the paths. Some paths have a number of tiles in common. For example, two paths can start the same way, but then at some tile, they split. Likewise, two paths that start separately might join up and end the same way (see the red an blue paths above). These would all be separate paths. The only reason to give these examples is to clarify what a path is.

(a) (5 points) Design and implement a recursive function called `marioCount()` that is given a and b (the size of the room, in tiles), and returns the total number of shortest paths Mario could take to get from the bottom left to the top right of the room.

(b) (3 points) Use your recursive function to calculate the number of paths for each of the following cases:

- 3×3
- 4×4
- 10×12

This can be done in your script; submit the output of these examples with the answers to the following questions.

(c) (5 points) For each base case in your function, briefly explain (a sentence or two) what the base case represents, and why the answer (which should be simple) is correct. It's okay if your function has several base cases, or only one.

(d) (5 points) For each recursive case in your function, briefly explain (2-5 sentences should do) how the problem size is made smaller, and how you build up the solution using the answer from a smaller problem size. It's okay if your function has several recursive cases, or only one.

Here are some hints:

- When you're thinking about this problem, it's natural to try to count the paths by finding all of them. That's fine, as you're trying to look for patterns, and for small values of a and b. But **your recursive function should not try to build any paths**; counting them is much simpler than finding them all.
- Use the recursion template.

What to Hand In

1. A file called `a7q7.py` containing: Your recursive `marioCount()` function and the code for the three examples listed part (b).
2. Add your explanations for parts (c) and (d) to the file called a7.txt (used for question 1-4). Clearly identify the question number and each part. You may also use RTF, DOC, DOCX, or PDF formats.

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

Evaluation

- (5 marks) Your `marioCount()` function is recursive, and correctly counts the number of paths.
- (3 marks) The output for the three test cases above is correct.
- (5 marks) Your explanation of the base case(s) conveys that you understood what the code is doing there.
- (5 marks) Your explanation of the recursive case(s) conveys that you understood what the code is doing there.

Non-credit activities

Here are some extra things you could do if you want to push yourself a bit.

1. There is a simple recursive program that solves the problem, which will get you full marks, but it's very slow when a and b get bigger. Give an argument for the time complexity of the simple program. Can you establish the space complexity, i.e., how much space it uses?
2. Find a way to make the simple program fast. In other words, find an algorithm whose worst case time complexity is much better than the simple program. Establish the time and space complexity of the improved algorithm.

3. Solve the problem without recursion.

None of this extra work gets you any marks. Doing the work is its own reward.