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 – Solutions and Grading

Algorithm Analysis and Recursion

Date Due: Wednesday, July 21, 11:59pm **Total Marks: 93**

Question 1 (4 points):

Purpose: Students will practice the following skills:

- Analyzing algorithms to assess their run time complexity.

Degree of Difficulty: Easy

References: You may wish to review the following:

- Chapter 18: Algorithm Analysis

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task overview

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 $O(\log_2 n)$ 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. In other words, do not assume they are equal, and do not assume they are constant. They could both change independently.

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 $O(m^2)$ steps? Note: treat m and n as independent input-size parameters.

What to hand in

In this assignment, you will include your submission for multiple questions in a single document named `a7.txt`. Make sure your name, student number, and NSID appear at the top of this document.

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: While students were not required to justify the answers, the solutions below briefly justify the answers, so that the solution can be more helpful for students who are trying to learn from their mistakes.

1. $O(n)$. The loop is repeated n times, and each repeat has $O(1)$ steps, and $n \times O(1) = O(n)$.
2. $O(n^2)$. The loop is repeated $n/2$ times, and each repeat has $O(\log_2 n)$ steps, and $n/2 \times O(\log_2 n) = O(n \log_2 n)$.
3. $O(m \log(n))$. The loop is repeated $\log(n)$ times, and each repeat has $O(m)$ steps, and $\log n \times O(m) = O(m \log n)$.
4. $O(m^2 n)$ The loop is repeated n times, and each repeat has $O(m^2)$ steps.

Notes to markers:

- One mark each, no justification needed.
- If the answer is correct and uses the big-O notation, give the mark.
- Don't try to explain the answer. Just check for correctness.
- It's okay for students to omit the base of any logarithm.

Question 2 (9 points):

Purpose: Students will practice the following skills:

- Analyzing algorithms to assess their run time complexity.

Degree of Difficulty: Easy

References: You may wish to review the following:

- Chapter 18: Algorithm Analysis

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task overview

The questions below present some nested loops for you to practice algorithm analysis. You will need to justify your work briefly. It's more important that you understand how to do this, than for you to get the right answers.

(a) (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(n):
2     j = 0
3     while j < n:
4         print(j - i)
5         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.

```
1 for i in range(len(alist)):
2     j = 0
3     while j < i:
4         alist[j] = alist[j] - alist[i]
5         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

In this assignment, you will include your submission for multiple questions in a single document named `a7.txt`. Make sure your name, student number, and NSID appear at the top of this document.

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 time complexity is $O(n^2)$.

   - Brief justification. The loop variables i and j are independent. The inner loop is $O(n)$, and the outer loop repeats $O(n)$ times. Thus we have $O(n) \times O(n) = O(n^2)$.
   - More detail: There are two loops, and the loop variables i and j are independent (the values that j can take do not depend on the values taken by i). 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 the inner loop does n iterations of $O(1)$ steps, so lines 3-5 are $O(n)$. Now, the outer loop repeats lines 3-5 n times, so a total of $n \times O(n) = O(n^2)$.
   - 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.

     ```
     xxxxxxxxxx
     xxxxxxxxxx
     xxxxxxxxxx
     xxxxxxxxxx
     xxxxxxxxxx
     xxxxxxxxxx
     xxxxxxxxxx
     xxxxxxxxxx
     xxxxxxxxxx
     xxxxxxxxxx
     ```

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

   - Brief Justification. This is a nested loop where the intter loop variable depends on the outer loop variable.

     Let n be the length of the list `alist`. 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 steps i from 0 to $n-1$. The inner loop steps j from 0 to i, so the inner loop is $O(i)$. The total is $O(n^2)$.
   - More detail. Let n be the length of the list `alist`; this is our input size parameter. 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 (lines 4-5) has 3 list index steps, two assignments, and a couple of arithmetic steps, so $O(1)$. The inner loop steps j from 0 to $i-1$. In other words, the body of the inner loop (lines 4-5) 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 is like tiling a triangle: i controls the rows and j controls the columns, and each $O(1)$ step is a tile. On row $i = 0$, there are no tiles. On row $i = 1$, there is 1 tile. On row i, there are i tiles. On row n there are n tiles. The triangle is roughly half of the square from the previous question.

```
..........
x.........
xx........
xxx.......
xxxx......
xxxxx.....
xxxxxx....
xxxxxxx...
xxxxxxxx..
xxxxxxxxx.
```

It's a triangle, but it's half the size of the square in part 1. And constant factors like "half" are discarded in asymptotic analysis. The area here is asymptotically the same as in part (a).

(The dots are there just to visualize the full square from part (a). Only the `x` represent the costs.)

3. The worst case complexity is $O(n \log_2 n)$

- Brief Justification. This is a logarithmic loop, nested within a linear loop. The inner loop steps j values as follows: 1, 2, 4, 8, ... THis can happen at most $O(\log_2 n)$ times. The body of the outer loop is repeated $O(n)$ times. The worst case time complexity is $O(n \log(n))$.
- More detail. 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 $j = 1$, with j doubling each iteration. Therefore the inner loop will repeat the body $\log_2(n)$ 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 "curves" to be more and more vertical as i increases.

```
xxx.......
xxx.......
xxx.......
xxx.......
xxx.......
xxx.......
xxx.......
xxx.......
xxx.......
```

The width of this rectangle is $O(\log_2 n)$. You basically have to double n to make the width larger by 1 `x`. The area here is asymptotically distinct from $O(n^2)$.

(The dots are there just to visualize the full square from part (a). Only the `x` represent the costs.)

Notes to markers: This question is about recognizing loop dependence or independence, and doing the right analysis.

- Each question is worth 3 marks.
- Part (a).
  - One mark for $O(n^2)$. Do not give the mark if big-O is not used, even if the answer says n^2.
  - 2 marks for the justification. Give the marks if the answer includes the idea that the loop variables are independent, so a simple multiply is appropriate.
- Part (b)
  - One mark for $O(n^2)$. Do not give the mark if big-O is not used, even if the answer says n^2.
  - 2 marks for the justification. Give the marks if the answer includes the idea that the loop variables are dependent. The work of the inner loop depends on the value for the outer loop variable.
  - The justification can use the formula, as shown above, but it is not required.
- Part (c)
  - One mark for $O(n \log_2 n)$. Do not give the mark if big-O is not used, even if the answer says $n \log_2 n$.
  - 2 marks for the justification. Give the marks if the answer includes the idea that the loop variables are independent. The answer also has to mention that the inner loop is a "logarithmic loop".
  - It's okay to leave out the base of the logarithm.

Question 3 (6 points):

Purpose: Students will practice the following skills:

- Analyzing algorithms to assess their run time complexity.

Degree of Difficulty: Easy

References: You may wish to review the following:

- Chapter 18: Algorithm Analysis

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

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
```

Note: The function `range()` is a class constructor for range objects. The behaviour is not really complicated, but it is hidden. When a range object is used with a for-loop (for example, Line 13), Python asks the range object for an integer, one at a time, so that the loop variable has the right values in the right order. All of this is "behind the scenes," so you can't observe it directly. For each value that the loop variable needs, the range object does a couple of steps of arithmetic. So the total cost of using the range object in a for-loop is $O(n)$, where n is the number of integers produced by the range.

However, used as a Boolean expression, the check `val in range(n)` can be done in $O(1)$ time. This is not obvious, either, and is the result of data stored inside the range object. Basically, the range object knows its limits; here they are `1` and `n+1`, and it can check if `val` is in the range, by checking the limits. This can be done in just a few arithmetic operations, and does not depend on `n`.

What to hand in

In this assignment, you will include your submission for multiple questions in a single document named `a7.txt`. Make sure your name, student number, and NSID appear at the top of this document.

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 following argument is plausible, and acceptable as correct, because of the trickiness of `range()`. But it's not accurate. See below!

- 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. However, we have implemented the LList ADT, and a good solution does not count the nodes when `size()` is called. It simply looks up the size determined by other operations. Python lists do the same thing.
- 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.
  - (This is the part where incorrect information is used.) 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 $O(n^3)$

Note: The above argument is plausible, but it's not the way Python works. A more accurate argument follows.

- 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.
  - (This is the part that corrects the analysis above.) The condition for Line 16 uses `range()` and the relational operator `in`. Normally, for lists and tuples, `in` uses linear search to look through the constructed range to find `val`. But in the context of a conditional, the check `if val in range(n)` can be done in $O(1)$ time, without using linear search. **This is not obvious, and is the result of data stored inside the `range()` object.** Basically, the range object knows its limits; here they are `1` and `n+1`, and it can ask if `val` is in the range, by checking the limits. This can be done in just a few operations, and does not depend on `n`. When we use range with a step value, e.g., `if val in range(0, n, 7)`, it's smart enough to check that `val` is one of the values that would be produced, without stepping through them. It boils down to a bit of arithmetic. The use of `not` adds a single step.
    So line 16 is $O(1)$.
  - The best case for the function is if the very first value in the very first row is not in the range. Since we've only looked at one value, and checking the range on line 16 is $O(1)$, be best case for the whole function is $O(1)$.
  - The worst case for the function would be if the function had to look at every value in the square. This would happen if the `if` condition were always `False`. That is, if every element in the square is in the right range.
  - 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 every element in the square is in the right range on line 16, and we go through all the values in the square. But checking the range on line 16 is still just $O(1)$, so we just have to do that for every value in the square.
- The worst case for the inner loop suggests we have to check the range for every value of j; that's $O(n)$ in total.
- The outer loop repeats the inner loop n times as well; the total cost for lines 13-17 is $n \times O(n) = O(n^2)$.
- The function is over on line 18. If the program got here, it completed the worst case for lines 13-17, $O(n^2)$.
- The worst-case complexity of the whole function is therefore $O(n^2)$

Notes to markers:

- 1 mark. Give the mark if the size parameter was correctly identified as the length of one side of the square. This is the length of the list `square`.
- 2 marks. Give the marks if the final conclusion is correct.
  - There are two answers, $O(n^3)$ and $O(n^2)$. Either one gets the marks.
- 3 marks. Analysis of the loops. The loop variables are independent. It is acceptable to take the cost of line 16 to be either $O(1)$ or $O(n)$, provided some explanation is given.
  - A good solution will identify the worst case for the function, and that comes from line 16 as well.
  - The best case analysis was given in the solutions above, but students were not required to provide the best case analysis.

Question 4 (15 points):

Purpose: Students will practice the following skills:

- Simple recursion on integers.

Degree of Difficulty: Easy

References: You may wish to review the following:

- Chapter 19: Recursion

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Note: To get the most value out of this question, practice using the techniques discussed in class. Consider using the template (*Chapter 19.2*) for recursive functions, and think about each question in terms of (a) the delegation metaphor, and (b) the relationship between the work of the delegates. You're not doing this question for the code that results. You are doing this question to practice designing recursive functions.

(a) 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 this:

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

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. Use an assertion to ensure that the argument is not negative!

(b) 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. Use an assertion to ensure that the argument is not negative!

(c) Design, implement, and test a function named `recsum(i, j)` that takes 2 integers as arguments, and returns the sum of the integers starting at `i` and up to but not including `j`. Informally, we might write it this way:

$$\text{recsum}(i, j) = i + (i + 1) + ... + (j - 1)$$

For example:

$$\text{recsum}(5, 10) = 5 + 6 + 7 + 8 + 9$$

Notice that i is included in the sum, but j is not. This is rather like the the way `range()` works: the first value is inclusive, but the second is exclusive.

Just to be clear, here are some quick test cases for you:

```
assert recsum(1, 0) == 0, 'Empty series'
assert recsum(0, 1) == 0, 'Series length 1'
assert recsum(0, 2) == 1, 'Series length 2'
assert recsum(0, 5) == sum(range(0, 5)), 'Series length 5'
assert recsum(5, 10) == sum(range(5, 10)), 'Series length 5, starting at 5'
```

Your function must be recursive. To avoid ambiguity, design your function so that, if $i \geq j$, then `recsum(i,j) == 0`. You can assume that your function will be called with integers; while your function might return a sensible answer when called with floating point values, it is not required that you design this function to be robust in this sense. We're studying recursion, and don't really need distractions.

(d) Design, implement, and test a function named `countoddrec(i, j)` that takes 2 integers as arguments, and returns the count of the number of odd integers starting at `i` and up to but not including `j`.

Just to be clear, here are some quick test cases for you:

```
assert countoddrec(1, 0) == 0, 'Empty sequence'
assert countoddrec(0, 1) == 0, 'Sequence length 1'
assert countoddrec(0, 2) == 1, 'Sequence length 2'
assert countoddrec(0, 5) == 2, 'Sequence length 4'
assert countoddrec(5, 10) == 3, 'Sequence length 5, starting at 5'
```

To avoid ambiguity, design your function so that, if $i \geq j$, then `countoddrec(i,j) == 0`. You can assume that your function will be called with integers; while your function might return a sensible answer when called with floating point values, it is not required that you design this function to be robust in this sense. We're studying recursion, and don't really need distractions.

Note: There is an easy calculation for this function, which requires no repetition or recursion. That's what you would use everywhere, except this assignment in which we are simply practicing recursion and design of recursive functions.

What to Hand In

- A Python program named `a7q4`.py containing your recursive functions.
- A Python script named `a7q4_testing.py` containing your test script. All of your functions should be well tested.

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 correct and recursive, zero marks otherwise.
- 2 marks: Moosonacci function. Full marks if it is correct and recursive, zero marks otherwise.
- 3 marks: `recsum(i, j)`. Full marks if it is correct and recursive, zero marks otherwise.
- 3 marks: `countoddrec(i, j)`. Full marks if it is correct and recursive, zero marks otherwise.
- 5 marks: Your functions are tested and have good coverage.

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)
```

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)
```

recsum
This function is slightly more interesting than the examples so far, but not by much.

```
def recsum(i, j):
    """
    Purpose:
        Calculate the sum of integers i through j (exclusive)
    Preconditions:
        :param i: a non-negative integer
        :param j: a non-negative integer
    Return:
        :return: the sum i + (i+1) + ... + (j - 1)
    """
    if i >= j:
        return 0
    else:
        return i + recsum(i + 1, j)
```

countoddrec
This function is a little interesting. A model solution is given below, though I do not expect this to be the solution submitted by most students.

```
def countoddrec(i, j):
    """
    Purpose:
        Calculate the number of odd integers in the
        range starting at i through j (exclusive)
    Preconditions:
        :param i: a non-negative integer
        :param j: a non-negative integer
    Return:
        :return: the number of odd integers in the range
    """
    if i >= j:
        return 0
    elif i % 2 == 1:
        return 1 + countoddrec(i + 2, j)
    else:
        return countoddrec(i + 1, j)
```

There are two recursive cases, one when i is odd, and one when i is even. Since we are counting odd numbers, we can skip from one odd number to another by adding 2 to i. I expect most submitted solutions will add 1 to i. This is acceptable, of course, but it is doing twice as much work!

Testing
A model solution can be found on Moodle. There are no surprises. I expect many students will have used the abbreviated format shown above, and that's totally acceptable. Better to have short test cases, than no testing at all! I still find the longer format more useful, when the length does not get in the way of presentation.

For recursive functions, we use white-box test cases. One test case for each base case in the function, and one or two test cases for each recursive case. Recursive functions are very narrowly focussed on one single task, and the test cases practically write themselves.

Notes for markers:

- Fibonacci: 2 marks. Full marks unless you see an error in the code.
- Moosonacci: 2 marks. Full marks unless you see an error in the code.
- recsum: 3 marks. Full marks unless you see an error in the code.
- countoddrec: 3 marks. Full marks unless you see an error in the code.
- Testing: 5 marks. Check that test cases cover base cases and recursive cases. Full marks unless you can see a deficiency somewhere.

  The test driver may imitate the "quick tests" shown in the question description above, with one single assertion per case. That's totally acceptable, and no deductions should be made for this kind of brevity.

Question 5 (17 points):

Purpose: Students will practice the following skills:

- Simple recursion on sequences like lists and strings.

Degree of Difficulty: Easy

References: You may wish to review the following:

- Chapter 19: Recursion

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Note: To get the most value out of this question, practice using the techniques discussed in class. Consider using the template (*Chapter 19.2*) for recursive functions, and think about each question in terms of (a) the delegation metaphor, and (b) the relationship between the work of the delegates. You're not doing this question for the code that results. You are doing this question to practice designing recursive functions.

In these questions we are working with lists and strings. For practice only, design functions that *make the list or string smaller using slices*. This is not efficient, because slicing creates new lists and strings, and creating them takes $O(n)$ time, when n is the length of the slice. But, for now, while we are building our skills, we will endure this inefficiency. We will correct it in a later question.

(a) Design, implement, and test a function named `recsumlist(alist)` that takes a list of numbers, and produces the sum of the numbers in the list. Your function must be recursive. To avoid any ambiguity, assume that the sum of an empty list is zero. Just to be clear, here are some quick test cases for you:

```
assert recsumlist([]) == 0, 'Empty list'
assert recsumlist([1, 3, 4]) == 8, 'Non-empty list'
```

Your function must be recursive. To avoid ambiguity, design your function so that the sum of an empty list is zero.

(b) Design, implement, and test a function named `recmemberlist(target, alist)` that takes a target value, and a list of values, and determines if the target value appears in the list.

Just to be clear, here are some quick test cases for you:

```
assert recmemberlist(0, []) == False, 'Empty list'
assert recmemberlist(3, [1, 3, 4]) == True, 'Non-empty list'
assert recmemberlist(5, [1, 3, 4]) == False, 'Non-empty list'
```

Your function must be recursive. To avoid any ambiguity, assume that the this function returns False for any target, given an empty list.

(c) Design, implement, and test a function named `reccountlist(target, alist)` that takes a target value, and a list of values, and counts the number of times the value appears in the list.

Just to be clear, here are some quick test cases for you:

```
assert reccountlist(0, []) == 0, 'Empty list'
assert reccountlist(3, [1, 3, 4, 3]) == 2, 'Non-empty list'
assert reccountlist(2, [1, 3, 4, 3]) == 0, 'Non-empty list'
```

Your function must be recursive. To avoid any ambiguity, assume that the this function returns 0 for any target, given an empty list.

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

```
assert subst_str('l', 'x', 'Hello, world!') == 'Hexxo, worxd!'
assert subst_str('o', 'i', 'Hello, world!') == 'Helli, wirld!'
assert subst_str('z', 'q', 'Hello, world!') == 'Hello, world!'
```

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 (i.e., it should return the correct value). If the target does not appear in the string, the returned string is identical to the original string.

(e) We need to practice thinking about algorithm analysis for recursive functions. As we said earlier, the run time cost of making a slice is $O(n)$ where n is the length of the resulting slice (not the original list). Using this information, determine the worst-case time complexity for your functions:

- `recsumlist()`
- `reccountlist()`
- `recmemberlist()`
- `subst_str()`

For each function, indicate what quantity you are using for input size, and express the the time complexity in terms of big-O. Remember that you have to count the number of function calls, taking into consideration any costs within each function call. Remember also the slicing has to included in this accounting.

Note: As already stated, slicing a list is $O(n)$, where n is the length of the resulting list or string. Another possibly surprising fact is that the addition operator has different complexity depending on how it is used. For numbers of every-day size, addition is $O(1)$, because the CPU has an add instruction than needs only one CPU step. However, for very, very big integers (with hundreds of decimal digits), addition is not $O(1)$, but $O(n)$, where n is the number of bits in the number. WE rrely have to worry about that.

However, when applied to strings and lists, the operator does not do arithmetic addition, but concatenation. Basically a new list or string needs to be created by copying the two component strings. So concatenation of lists or strings using the $+$ operator is $O(n)$, where n is the length of the resulting string or list.

What to Hand In

- A Python program named `a7q5.py` containing your recursive functions.
- A Python script named `a7q5_testing.py` containing your test script. All of your functions should be well tested.
- A text document named `a7q5.txt` containing your ~~recursive functions~~ algorithm analysis for a7q5(e). You can also use DOC, DOCX, PDF or RTF formats for this document.

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

Evaluation

- 2 marks: `recsumlist(alist)`. Full marks if it is recursive, zero marks otherwise.
- 2 marks: `recmemberlist(target, alist)`. Full marks if it is recursive, zero marks otherwise.
- 2 marks: `reccountlist(target, alist)`. Full marks if it is recursive, and if it works, zero marks otherwise.
- 2 marks: `subst_str(t, r, s)`. Full marks if it is recursive, and if it works, zero marks otherwise.
- 5 marks: Your functions are tested and have good coverage.
- 4 marks (1 mark each): your algorithm analysis results for these functions is correct.

Solution:

recsumlist

The following function is pretty straight-forward.

```
def recsumlist(alist):
    """
    Purpose:
        Return the sum of the values in the list
    Preconditions:
        :param alist: a list containing numbers
    Return:
        :return: the sum of the values in the list
    """
    if len(alist) == 0:
        return 0
    else:
        return alist[0] + recsumlist(alist[1:])
```

Analysis.

- The Python functionlen() is $O(1)$ on Python lists (because the length is stored in the list object, the same way that we stored the size of our LList objects in A6).
- Our size parameter will be N, which we use to represent the size of the initial list. This is distinct from n, the length of the list as we keep working on it; n keeps changing, but N doesn't.
- In the recursive case, list indexing and addition are both 1 step, but slicing is $O(n)$, where n is the length of the slice.
- The list keeps getting smaller by one element on each recursive call, so we call the function $O(N)$ times.
- Each time we call the function, the list is a little smaller. So the work on each recursive cll gets a little smaller as well.
- The total amount of work is:

$$O(N) + O(N-1) + O(N-2) + \cdots + O(1) = O(N^2)$$

- A simpler analysis is acceptable: We have N function calls, and each call has to slice the list, which is $O(N)$, so multiplying gives $O(N^2)$. This is a little imprecise, but it is not an incorrect argument, asymptotically.
- A fully precise analysis requires math that is taught in CMPT 260, so we will not insist on a precise argument here.

recmemberlist
This function is very similar to the previous. We need two base cases. The first base case looks like it applies to a very special case, but whenever the value we're looking for is not in the list, we get to it. The second base case also looks like a very special case, but it's the case that we get to eventually when the value we are looking for is present in the list. The recursive case basically says "keep looking." The answer to the delegation is passed on. There is no need to do extra work, unlike `recsumlist()` where we had to do some addition after the answer was obtained.

```
def recmemberlist(value, alist):
    """
    Purpose:
        Check if the given value appears in the given list.
    Preconditions:
        :param value: a value
        :param alist: a list of values
    Return:
        :return: True if the value appears in the list, False otherwise
    """
    if len(alist) == 0:
        return False
    elif alist[0] == value:
        return True
    else:
        return recmemberlist(value, alist[1:])
```

Analysis.

- Our size parameter will be N, which we use to represent the size of the initial list. This is distinct from n, the length of the list as we keep working on it; n keeps changing, but N doesn't.
- The list keeps getting smaller by 1 each time the function is called.
- The only costly step in the function is the slicing in the recursive case, and it's $O(n)$, where n is the size of the slice.
- In the worst case, the value is not in the list at all, and we have to go all the way to the end to prove it. The total amount of work is:

  $$O(N) + O(N-1) + O(N-2) + \cdots + O(1) = O(N^2)$$

  This is for the worst case. This is considerably worse than a loop that walks through the list!
- A best case is possible, namely, the item appears at the very front of the original list. It can be fount without any slicing or recursion. The best case time complexity is therefore $O(1)$.

reccountlist
This function is slightly more elaborate than the previous. Because we have to count, one of the base cases in the previous function has to be a recursive case here. And because we are counting, the recursive cases need to be do a bit more work.

```
def reccountlist(value, alist):
    """
    Purpose:
        Count the number of times the given value appears in the given list.
    Preconditions:
        :param value: a value
        :param alist: a list of values
    Return:
        :return: the number of times the value appears in the list
    """
    if len(alist) == 0:
        return 0
    elif alist[0] == value:
        return 1 + reccountlist(value, alist[1:])
    else:
        return reccountlist(value, alist[1:])
```

Analysis.

- Our size parameter will be N, which we use to represent the size of the initial list. This is distinct from n, the length of the list as we keep working on it; n keeps changing, but N doesn't.
- The list keeps getting smaller by 1 each time the function is called.
- The only costly step in the function is the slicing in the recursive case, and it's $O(n)$, where n is the size of the slice.
- In this function, we have to go all the way to the end to do a full accounting, so there is no difference between worst and best cases, unlike the previous task. The total amount of work is due to slicing a smaller and smaller list:

$$O(N) + O(N-1) + O(N-2) + \cdots + O(1) = O(N^2)$$

subst_str

Here we have an example of constructing an answer that's not a number. Other than that, there is nothing exciting here.

```
def subst_str(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 + subst_str(t, r, s[1:])
    else:
        return s[0] + subst_str(t, r, s[1:])
```

Analysis

- Our size parameter will be N, which we use to represent the size of the initial string. This is distinct from n, the length of the string as we keep working on it; n keeps changing, but N doesn't.
- The string keeps getting smaller by 1 each time the function is called.
- Slicing a string requires $O(n)$ steps, just as for a list.
- The operation + for strings is also $O(n)$, where n is the size of the resulting string. To understand this, we have to realize that a new string object is being created, and that all the characters in both strings have to be copied to the new string. The characters are copied one at a time, so $O(n)$.
- Each recursive case has $O(n)$ to create the new string, and $O(n)$ to slice the original string, so the total on each recursive call is $O(n)$.
- The total amount of work is due to slicing a smaller and smaller list:

$$O(N) + O(N-1) + O(N-2) + \cdots + O(1) = O(N^2)$$

Notes for markers:

- `recsumlist()`
  - 2 marks. Full marks unless you see an error in the code.
- `recsmemberist()`
  - 2 marks. Full marks unless you see an error in the code.
- `reccountlist()`
  - 2 marks. Full marks unless you see an error in the code.
- `subst_str()`
  - 2 marks. Full marks unless you see an error in the code.
- Algorithm analysis.
  - 4 marks, 1 mark each. To get the mark, the conclusion has to be right. It's all $O(N^2)$. Deduct the mark if you can detect a very bad analysis. Otherwise, be generous.
- Testing. 5 marks. Check that test cases cover base cases and recursive cases. Full marks unless you can see a deficiency somewhere.

  The test driver may imitate the "quick tests" shown in the question description above, with one single assertion per case. That?s totally acceptable, and no deductions should be made for this kind of brevity.

Question 6 (15 points):

Purpose: Students will practice the following skills:

- Simple recursion on sequences like lists and strings.

Degree of Difficulty: Easy

References: You may wish to review the following:

- Chapter 19: Recursion

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Note: To get the most value out of this question, practice using the techniques discussed in class. Consider using the template (*Chapter 19.2*) for recursive functions, and think about each question in terms of (a) the delegation metaphor, and (b) the relationship between the work of the delegates. You're not doing this question for the code that results. You are doing this question to practice designing recursive functions.

In these questions we are working with lists and strings. In the previous question, we implemented the recursion by making the lists smaller using slices, and we mentioned that this technique is inefficient. In this question, we will fix that problem.

In the following functions, you will use an integer index as an extra parameter, so that you can walk the index from 0 up to the length of the list or string, and avoid slices. You will know where you are in the list by using the index; the index will change, but the list will not. **Your recursion must be based on a changing index, not a changing list.**

(a) Design, implement, and test a function named `recsumlisti(index, alist)` that takes an integer index, and a list of numbers, and produces the sum of the numbers in the list from the given index.

Just to be clear, here are some quick test cases for you:

```
assert recsumlisti(0, []) == 0, 'Empty list'
assert recsumlisti(0, [1, 3, 4]) == 8, 'start at index 0'
assert recsumlisti(1, [1, 3, 4]) == 7, 'start at index 1'
assert recsumlisti(2, [1, 3, 4]) == 4, 'start at index 2'
assert recsumlisti(3, [1, 3, 4]) == 0, 'start at index 3'
```

Your function must be recursive. To avoid ambiguity, design your function so that the sum of an empty list is zero, and if the index is too big, the sum is also zero.

(b) Design, implement, and test a function named `recmemberlisti(index, target, alist)` that takes an integer index, a target value, and a list of values, and determines if the target value appears in the list starting at the given index.

Just to be clear, here are some quick test cases for you:

```
assert recmemberlisti(0, 0, []) == False, 'Empty list'
assert recmemberlisti(0, 3, [1, 3, 4]) == True, 'start at index 0'
assert recmemberlisti(1, 3, [1, 3, 4]) == True, 'start at index 1'
assert recmemberlisti(2, 3, [1, 3, 4]) == False, 'start at index 2'
```

Your function must be recursive. To avoid any ambiguity, assume that the this function returns False for any target, given an empty list, and False if the index is too big for the list.

(c) Design, implement, and test a function named `reccountlisti(index, target, alist)` that takes an integer index, a target value, and a list of values, and counts the number of times the value appears in the list at the given index, or later.

Just to be clear, here are some quick test cases for you:

```
assert reccountlisti(0, 0, []) == 0, 'Empty list'
assert reccountlisti(0, 3, [1, 3, 4, 3]) == 2, 'start at index 0'
assert reccountlisti(1, 3, [1, 3, 4, 3]) == 2, 'start at index 1'
assert reccountlisti(2, 3, [1, 3, 4, 3]) == 1, 'start at index 2'
```

Your function must be recursive. To avoid any ambiguity, assume that the this function returns 0 for any target, given an empty list, and ~~False~~ 0 if the index is too big.

(d) We need to practice thinking about algorithm analysis for recursive functions. Determine the worst-case time complexity for your functions:

- `recsumlisti()`
- `recmemberlisti()`
- `reccountlisti()`

For each function, indicate what quantity you are using for input size, and express the the time complexity in terms of big-O. Remember that you have to count the number of function calls, taking into consideration any costs within each function call. If you have implemented the functions correctly, you will not be using slicing here, and as a result, your analysis should show that these functions are more efficient than the previous question's versions!

(e) You may have noticed that we did not include `subst_str()` in this question. Can the technique of using an integer index be used in `subst_str()` to improve the runtime costs? Why or why not? Be careful here. The answer is not in the recursion, or the slicing, but the nature of strings!

All of your functions should be well tested.

What to Hand In

- A Python program named `a7q6.py` containing your recursive functions.
- A Python script named `a7q6_testing.py` containing your test script.
- A text document named `a7q6.txt` containing your algorithm analysis for a7q6(d) and a7q6(e). You can also use DOC, DOCX, PDF or RTF formats for this document.

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

Evaluation

If any of your functions use slicing, you will receive a zero for that function. You must use recursion based on the changing index.

- 2 marks: `recsumlisti(alist)`. Full marks if it is recursive and correct, zero marks otherwise.
- 2 marks: `recmemberlisti(target, alist)`. Full marks if it is recursive and correct, zero marks otherwise.
- 2 marks: `reccountlisti(target, alist)`. Full marks if it is recursive and correct, zero marks otherwise.
- 5 marks: Your functions are tested and have good coverage.
- 3 marks (1 mark each): your algorithm analysis results for these functions is correct.

- 1 mark: you correctly explained why the indexing technique cannot improve the complexity of the `subst_str()` from Question 5.

Solution: Hey slicer! No slicing!

recsumlisti

When we use an integer index, we do not modify the list at all. The only problematic aspect of this implementation is that we do not have control of the initial index. To look at the whole list, the initial index should be 0. We have to trust that the correct initial value is used.

```
def recsumlisti(index, alist):
    """
    Purpose:
        Return the sum of the values in the list, starting at index
    Preconditions:
        :param index: a valid integer index
        :param alist: a list containing numbers
    Return:
        :return: the sum of the values in the list
    """
    if index >= len(alist):
        return 0
    else:
        return alist[index] + recsumlisti(index + 1, alist)
```

Analysis.

- Our size parameter will be N, which we use to represent the size of the list.
- The Python functionlen() is $O(1)$ on Python lists (because the length is stored in the list object, the same way that we stored the size of our LList objects in A6).
- The base case is $O(1)$.
- The recursive case is $O(1)$, because addition and indexing are both single steps.
- The function has to be called N times, to see every value in the list.
- The total amount of work is $N \times O(1) = O(N)$.
- A slightly more precise analysis would look at the difference between N and the first index. It would not change the result, but it would account for use cases where the initial value for the index is not 0.

recmemberlisti
This function is very similar to the previous. It's also similar to the version in the previous question.

```
def recmemberlisti(index, value, alist):
    """
    Purpose:
        Check if the given value appears in the given list
        starting at index.
    Preconditions:
        :param index: a valid integer index
        :param value: a value
        :param alist: a list of values
    Return:
        :return: True if the value appears in the list, False otherwise
    """
    if index >= len(alist):
        return False
    elif alist[index] == value:
        return True
    else:
        return recmemberlisti(index + 1, value, alist)
```

Analysis.

- Our size parameter will be N, which we use to represent the size of the list.
- The Python functionlen() is $O(1)$ on Python lists (because the length is stored in the list object, the same way that we stored the size of our LList objects in A6).
- The base case is $O(1)$ because indexing is a single step.
- The recursive case is $O(1)$, because addition is a single step.
- In the worst case, (i.e., the value is not in the list), the function has to be called N times, to see every value in the list. The total amount of work is $N \times O(1) = O(N)$.
- A best case is possible, namely, the item appears at the very front of the original list. It can be fount without any additional function calls. The best case time complexity is therefore $O(1)$.

reccountlisti
By now, I expect that students were finding the patterns familiar and not too difficult.

```
def reccountlisti(index, value, alist):
    """
    Purpose:
        Count the number of times the given value appears in the given list.
    Preconditions:
        :param index: a valid integer index
        :param value: a value
        :param alist: a list of values
    Return:
        :return: the number of times the value appears in the list
    """
    if index >= len(alist):
        return 0
    elif alist[index] == value:
        return 1 + reccountlisti(index + 1, value, alist)
    else:
        return reccountlisti(index + 1, value, alist)
```

Analysis.

- Our size parameter will be N, which we use to represent the size of the list.
- The Python functionlen() is $O(1)$ on Python lists (because the length is stored in the list object, the same way that we stored the size of our LList objects in A6).
- The base case is $O(1)$ because indexing is a single step.
- The recursive cases are both $O(1)$, because addition is a single step.
- The function has to be called N times, to see every value in the list. The total amount of work is $N \times O(1) = O(N)$.

subst_str
There was no code required here. However, there was a question about why this was not included. The answer is that we could of course use the same technique, but because of the nature of Python strings and their operations, we cannot hope to improve the complexity. We can walk down a string using the indexing technique, but we cannot mutate a string, so we will have to create a new string with the letters we want changed. AS we saw in the previous question, creating a new string means copying the string, and if we make a new string N times, the total cost will be the same: $O(N^2)$. Another way to say it is that the indexing technique removed one of the two expensive computations in Q4, but it did not remove the other.

Note: Python gives us a string operation that is essentially the same as `subst_str()`, but since it can access the hidden data directly, the complexity is $O(N)$.

Notes for markers:

- `recsumlisti()`
  - 2 marks. Full marks unless you see an error in the code.
- `recsmemberisti()`
  - 2 marks. Full marks unless you see an error in the code.
- `reccountlisti()`
  - 2 marks. Full marks unless you see an error in the code.
- Algorithm analysis.
  - 3 marks, 1 mark each. To get the mark, the conclusion has to be right. It's all $O(N)$. Deduct the mark if you can detect a very bad analysis. Otherwise, be generous.
- 1 mark. Discussion of `subst_str()`. Give a mark for any explanation that includes the idea that strings still need to be copied every time the function is called. Be generous. It was a bit tricky.
- Testing. 5 marks. Check that test cases cover base cases and recursive cases. Full marks unless you can see a deficiency somewhere.

  The test driver may imitate the "quick tests" shown in the question description above, with one single assertion per case. That?s totally acceptable, and no deductions should be made for this kind of brevity.

Question 7 (14 points):

Purpose: Students will practice the following skills:

- Simple recursion on seqnode-chains.

Degree of Difficulty: Moderate

References: You may wish to review the following:

- Chapter 19: Recursion

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task overview

In preparation for our up-coming unit on trees, where recursive functions are the only option, we will practice writing recursive functions using node chains. Note that the node ADT is recursively defined, since the `next` field refers to another node-chain (possibly empty). We are practicing recursion in using a familiar ADT, so that when we change to a new ADT, we will have some experience.

Below are three exercises that ask for recursive functions that work on node-chains (**not Linked Lists, and not Python lists**). You MUST implement them using the node ADT (given), and you MUST use recursion (even though there are other ways). We will impose very strict rules on implementing these functions which will benefit your understanding of our upcoming work on trees.

For ~~one of the~~ these questions you are **not** allowed to use any data collections (lists, stacks, queues). Instead, recursively pass any needed information as arguments. **Do not add any extra parameters.** None are needed. Learn to work withing the constraints, because you will need these skills!

You will implement the following functions:

(a) Design, implement, and test a function named `sumnc_rec(chain)` that takes a node-chain containing numbers as data, and produces the sum of the numbers in the chain. Your function must be recursive. To avoid any ambiguity, assume that the sum of an empty node-chain is zero.

(b) Design, implement, and test a function named `membernc_rec(target, chain)` that takes a target value, and a node-chain of data values, and determines if the target value appears in the node-chain. Your function must be recursive. To avoid any ambiguity, assume that the this function returns False for any target, given an empty node-chain.

(c) Design, implement, and test a function named `countnc_rec(chain, target)` that takes a target value, and a node-chain of data values, and counts the number of times the value appears in the node-chain. Your function must be recursive. To avoid any ambiguity, assume that the this function returns 0 for any target, given an empty node-chain.

(d) We need to practice thinking about algorithm analysis for recursive functions. Determine the worst-case time complexity for your functions:

- `sumnc_rec()`
- `countnc_rec()`
- `membernc_rec()`

For each function, indicate what quantity you are using for input size, and express the the time complexity in terms of big-O. Remember that you have to count the number of function calls, taking into consideration any costs within each function call.

What to Hand In

- A Python program named a7q7.py containing your recursive functions described above.
- A Python script named `a7q7_testing.py`; include the cases above and tests you consider important.
- A text document named `a7q7.txt` containing your algorithm analysis for a7q7(d). You can also use DOC, DOCX, PDF or RTF formats for this document.

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

Evaluation

- 2 marks: `sumnc_rec(chain)`. Full marks if it is recursive, zero marks otherwise.
- 2 marks: `membernc_rec(target, chain)`. Full marks if it is recursive, zero marks otherwise.
- 2 marks: `countnc_rec(chain, target)`. Full marks if it is recursive, and if it works, zero marks otherwise.
- 5 marks: Your functions are tested and have good coverage.
- 3 marks (1 mark each): your algorithm analysis results for these functions is correct.

Solution:

sumnc_rec

When we use recursion on node chains, the functions are pretty simple.

```
def sumnc_rec(chain):
    """
    Purpose:
        Return the sum of the values in the
    Preconditions:
        :param chain: a node-chain containing numbers
    Return:
        :return: the sum of the values in the node-chain
    """
    if chain is None:
        return 0
    else:
        return chain.data + sumnc_rec(chain.next)
```

Analysis.

- Our size parameter will be N, which we use to represent the size of the chain.
- The base case is $O(1)$.
- The recursive case is $O(1)$, because addition and accessing an attribute are both single steps.
- The function has to be called N times, to see every value in the list.
- The total amount of work is $N \times O(1) = O(N)$.

Node chains do not requiring slicing, and there's no need to use the indexing technique of the previous question. It simply can't get any simpler.

membernc_rec
This function is very similar to the previous. It's also similar to the version in the previous question.

```
def membernc_rec(value, chain):
    """
    Purpose:
        Check if the given value appears in the given node-chain
    Preconditions:
        :param value: a value
        :param chain: a node-chain of values
    Return:
        :return: True if the value appears in the node-chain, False otherwise
    """
    if chain is None:
        return False
    elif chain.data == value:
        return True
    else:
        return membernc_rec(value, chain.next)
```

Analysis.

- Our size parameter will be N, which we use to represent the size of the chain.
- The base case is $O(1)$.
- The recursive cases are both $O(1)$.
- In the worst case, (i.e., the value is not in the chain), the function has to be called N times, to see every value. The total amount of work is $N \times O(1) = O(N)$.
- A best case is possible, namely, the item appears at the very front of the chain. It can be fount without any additional function calls. The best case time complexity is therefore $O(1)$.

countnc_rec
This is just more practice. Once the patterns are recognized, the execution is straight-forward.

```
def countnc_rec(value, chain):
    """
    Purpose:
        Count the number of times the given value appears in the given node-chain.
    Preconditions:
        :param value: a value
        :param chain: a node-chain of values
    Return:
        :return: the number of times the value appears in the node-chain
    """
    if chain is None:
        return 0
    elif chain.data == value:
        return 1 + countnc_rec(value, chain.next)
    else:
        return countnc_rec(value, chain.next)
```

Analysis.

- Our size parameter will be N, which we use to represent the size of the list.
- The base case is $O(1)$.
- The recursive cases are both $O(1)$.
- The function has to be called N times, to see every value in the list. The total amount of work is $N \times O(1) = O(N)$.

Notes for markers:

- `sumnc_rec()`
  - 2 marks. Full marks unless you see an error in the code.
- `membernc_rec()`
  - 2 marks. Full marks unless you see an error in the code.
- `countnc_rec()`
  - 2 marks. Full marks unless you see an error in the code.
- Algorithm analysis.
  - 3 marks, 1 mark each. To get the mark, the conclusion has to be right. It's all $O(N)$. Deduct the mark if you can detect a very bad analysis. Otherwise, be generous.
- Testing. 5 marks. Check that test cases cover base cases and recursive cases. Full marks unless you can see a deficiency somewhere.

  The test driver may imitate the "quick tests" shown in the question description above, with one single assertion per case. That?s totally acceptable, and no deductions should be made for this kind of brevity.

Question 8 (13 points):

Purpose: Students will practice the following skills:

- Simple recursion on seqnode-chains.

Degree of Difficulty: Moderate

References: You may wish to review the following:

- Chapter 19: Recursion

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task overview

In preparation for our up-coming unit on trees, where recursive functions are the only option, we will practice writing recursive functions using node chains. Note that the node ADT is recursively defined, since the `next` field refers to another node-chain (possibly empty). We are practicing recursion in using a familiar ADT, so that when we change to a new ADT, we will have some experience.

Below are three exercises that ask for recursive functions that work on node-chains (**not Linked Lists, and not Python lists**). You MUST implement them using the node ADT (given), and you MUST use recursion (even though there are other ways). We will impose very strict rules on implementing these functions which will benefit your understanding of our upcoming work on trees.

For ~~one of the~~ these questions you are **not** allowed to use any data collections (lists, stacks, queues). Instead, recursively pass any needed information as arguments. **Do not add any extra parameters.** None are needed. Learn to work withing the constraints, because you will need ths skills!

You will implement the following functions:

(a) `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`.

(b) In Assignment 5, Question 2, we defined a function called `check_chains(chain1, chain2)`. Its purpose was to examine 2 node-chains, and determine if they contained the same data. In this question, we're going to deal with a slightly simpler, but related, task.

- `check_chains(chain1, chain2)` will return `True` if they have the same data values in the same order.
- `check_chains(chain1, chain2)` will return `False` if there is any difference in the data values.

You do not need to return the index where the two chains differ. This is supposed to be a simpler task!

(c) `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.

Note: Only a *shallow* copy is required; if data stored in the original node chain is mutable, it does not also need to be copied.

(d) `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.

What to Hand In

- A Python program named a7q8.py containing your recursive functions described above.
- A Python script named `a7q8_testing.py`; include the cases above and 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

- 2 marks: `to_string(node_chain)`. Full marks if it is recursive, zero marks otherwise.
- 2 marks: `check_chains(chain1, chain2)`. Full marks if it is recursive, zero marks otherwise.
- 2 marks: `copy(node_chain)`. Full marks if it is recursive, zero marks otherwise.
- 2 marks: `replace(node_chain, target, replacement)`. Full marks if it is recursive, and if it works, zero marks otherwise.
- 5 marks: Your functions are tested and have good coverage.

Solution:

Students are not allowed to use any LISTS or data collections. This is to get them thinking about recursion in new ways, and consider how parameters and return statements can be used. Dock them 2 marks for each question that they used a list with.

to_string()

`to_string(node_chain)` requires two base cases. The first is when the node-chain is empty to start. The second one is the base case we get to if the node-chain is not empty. This is very similar to the multiple base cases for Fibonacci in Q4.

The recursive case is a little interesting, but not because of the recursion. Notice how the delegate's answer is combined with the delegator's information using the `format()` method. It could have been done using string concatenation as well; there is no technical reason to choose one over the other. I chose this one simply to show something a little different.

Students are not allowed to create any new parameters for this one. Any recursive variation is allowable.

```
def to_string(node_chain):
    """
    Purpose:
        Create a string representation of the node chain.  E.g.,
        [ 1 | *-]-->[ 2 | *-]-->[ 3 | / ]
    Pre-conditions:
        :param node_chain:  A node-chain, possibly empty
    Post_conditions:
        None
    Return: A string representation of the nodes.
    """
    # special case, empty node-chain
    if node_chain is None:
        return "EMPTY"
    elif node_chain.next is None:
        return "[ {} | / ]".format(node_chain.data)
    else:
        return "[ {} | *-]-->{}".format(node_chain.data,
                                        to_string(node_chain.next))
```

check_chains() Here, we just walk along both chains, taking one step at a time, checking for differences. The first base case checks if the references are the same, and if so, the chains must be equal. The second base case checks if one of the two is empty (they cannot both be empty, because that would have been determined in the first base case). If one is empty, the chains are different. The third base case checks the data value at the front of the chains; if different, they are different. The recursive case checks the rest of the chain after the first node. If we get this far, it's because the nodes contain the same data value. WE can't stop here, as we need to check the rest of the chains. However, the answer we get, either True or False is the final answer. There is no need to see what the answer is, or to do any kind of combination. The delegate's answer is identical to our answer.

```
def check_chains(chain1, chain2):
    """
    Purpose:
        Check if the values in the two chains are equal.
    Pre-Conditions:
        :param chain1: a node-chain, possibly empty
        :param chain2: a node-chain, possibly empty
    Post-conditions:
        (none)
    Return:
        :return: True if the two chains have the same values in the
                 same order
    """
    if chain1 is chain2:
        return True
    elif chain1 is None or chain2 is None:
        return False
    elif chain1.data != chain2.data:
        return False
    else:
        return check_chains(chain1.next, chain2.next)
```

copy()

Eventually, even recursion gets boring. The only fun thing here is how simple it is.

```
def copy(node_chain):
    """
    Purpose
        Make a completely new copy of the given node-chain.
    Preconditions:
        :param node_chain: a node-chain
    Post-conditions:
        None
    Return:
        :return: A new copy of the node-chain is returned
    """
    if node_chain is None:
        return None
    else:
        return N.node(node_chain.data, copy(node_chain.next))
```

replace()

The replace function reuses the nodes, and simply changes the data stored. We have included an additional parameter in the solution that keeps track of the current node in the node-chain

```
def replace(node_chain, target, replacement):
    """
    Purpose:
        Replace every occurrence of data target in node_chain with the data
        in replacement. The chain's data may change, but the structure will not.
    Pre-Conditions:
        :param node_chain: a node-chain, possibly empty
        :param target: a data value
        :param replacement: a data value
    Post-conditions:
        The node-chain is changed, by replacing target with value everywhere.
    Return:
        :return: None
    """
    # base case: we are at the end of the chain
    if node_chain is None:
        return None
    else:
       # if the current node contains data that should be replaced
       # replace it
       if node_chain.data == target:
           node_chain.data = replacement

       # look for more, no matter what
       replace(node_chain.next, target, replacement)

       return node_chain
```

Notes for markers:

- `to_string()`
  - 2 marks. Full marks unless you see an error in the code.
- `check_chains()`
  - 2 marks. Full marks unless you see an error in the code.
- `copy()`
  - 2 marks. Full marks unless you see an error in the code.
- `replace()`
  - 2 marks. Full marks unless you see an error in the code.
- Testing. 5 marks. Check that test cases cover base cases and recursive cases. Full marks unless you can see a deficiency somewhere.

  The test driver may imitate the “quick tests” shown in the question description above, with one single assertion per case. That?s totally acceptable, and no deductions should be made for this kind of brevity.