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: Monday, July 18, 11:59pm **Total Marks: 93**

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.
- This assignment is homework assigned to students and will be graded. This assignment shall not be distributed, in whole or in part, 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 assignment 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.**
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to com-pare every pair of student submissions. Plagiarism can include: copying answers from a web page, or from a classmate, or from solutions published in previous semesters. Basically, if you cannot delete your whole assignment and do it again yourself (given adequate time), it's not your work, so don't try to claim credit for it. Your success in this course depends on what you can do, not on what you can hand in.
- 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.
- Read the purpose of each question. Read the Evaluation section of each question.
- 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. It might help you, but it adds an extra step for the markers.
- Programs must be written in Python 3.
- **Assignments must be submitted to Canvas.** If you are not sure, talk to a Lab TA about how to do this.
- **Canvas 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. **Do not send late assignment submissions to your instructors, lab TAs, or markers. If you require an extension, request it in advance using email.**

Version History

- **07/19/2021**: Explained the runtime costs for concatenation for Question 5. Explained the runtime costs involved using the range object in Question 3.
- **07/18/2021**: Clarified how to hand in the algorithm analysis for questions 5-7.
- **07/15/2021**: Some minor corrections:
  - Questions 1-3: Fixed a typo making the "What to hand in" section ambiguous. Please put all your answers to Q1-Q3 in `a7.txt`.
  - Made very minor corrections to typos and removed redundancies in several questions.
  - A7Q6c. The function `reccountlisti(index, target, alist)` should return 0 if the index is too big.
  - A7Q6e. Added one point to this question giving a mark for A7Q6e.
- **07/14/2021**: released to students

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.

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.

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.

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.

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.

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: `recsumlist(alist)`. Full marks if it is recursive and correct, zero marks otherwise.
- 2 marks: `recmemberlist(target, alist)`. Full marks if it is recursive and correct, zero marks otherwise.
- 2 marks: `reccountlist(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.

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.

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.