NYU, Tandon School of Engineering

CS-UY 1134 Data Structures and Algorithms — Fall 2020

Homework #4

Submission instructions:

1. In all questions, unless explicitly stated otherwise, you are **not allowed** to:
    a. Define a helper function.
    b. Add parameters to the function's header line
    c. Set default values to the parameters
    d. Use global variables

2. You should turn in 7 (or 8) files:
    - A '.pdf' file with your written answers to questions 1 and 2.
      Name your file: 'YourNetID_hw4.pdf'
    - 6 (or 7) '.py' files: one, with all the code related, to each of the questions 3-9.
      Name your files: 'YourNetID_hw4_q3.py', 'YourNetID_hw4_q4.py', etc.

    Note: your netID follows an abc123 pattern, not N12345678.

3. You should submit your homework via Gradescope. For Gradescope's autograding feature to work:
    a. Name all classes, functions and methods **exactly as they are in the assignment specifications**.
    b. Make sure there are **no print statements** in your code. If you have tester code, please put it in a "main" function and **do not call it**.

Question 1:

You are given 2 implementations for a recursive algorithm that calculates the sum of all the elements in a list (of integers):

```
def sum_lst1(lst):
  if (len(lst) == 1):
      return lst[0]
  else:
      rest = sum_lst1(lst[1:])
      sum = lst[0] + rest
      return sum


def sum_lst2(lst, low, high):
  if (low == high):
      return lst[low]
  else:
      rest = sum_lst2(lst, low + 1, high)
      sum = lst[low] + rest
      return sum
```

Note: The implementations differ in the parameters we pass to these functions:

- In the first version we pass only the list (all the elements in the list have to be taken in to account for the result).
- In the second version, in addition to the list, we pass two indices: low and high (low ≤ high), which indicate the range of indices of the elements that should to be considered.
  The initial values (for the first call) passed to low and high would represent the range of the entire list.

1) Make sure you understand the recursive idea of each implementation.
2) Analyze the running time of the implementations above. For each version:
    i) Draw the recursion tree that represents the execution process of the function, and the local-cost of each call.
    ii) Conclude the total (asymptotic) running time of the function.
3) Which version is asymptotically faster?

Question 2:

Analyze the running time of each of the following functions. For each function:

i. Draw the recursion tree that represents the execution process, and the cost of each call.
ii. Conclude the total (asymptotic) running time.

Note: For the simplicity of the analysis of sections (b) and (c), you may assume that n is a power of 2, therefore it can always be divided evenly by 2.

a.

```
def fun1(n):
  if (n == 0):
      return 1
  else:
      part1 = fun1(n-1)
      part2 = fun1(n-1)
      res = part1 + part2
      return res
```

b.

```
def fun2(n):
  if (n == 0):
      return 1
  else:
      res = fun2(n//2)
      res += n
      return res
```

c.

```
def fun3(n):
  if (n == 0):
      return 1
  else:
      res = fun3(n//2)
      for i in range(1, n+1):
          res += i
      return res
```

Question 3:

Give a **recursive** implement to the following functions:

a. **def** `print_triangle(n)`

This function is given a positive integer `n`, and prints a textual image of a right triangle (aligned to the left) made of *n* lines with asterisks.

For example, `print_triangle(4)`, should print:

```
*
**
***
****
```

b. **def** `print_oposite_triangles(n)`

This function is given a positive integer `n`, and prints a textual image of a two opposite right triangles (aligned to the left) with asterisks, each containing *n* lines.

For example, `print_oposite_triangles(4)`, should print:

```
****
***
**
*
*
**
***
****
```

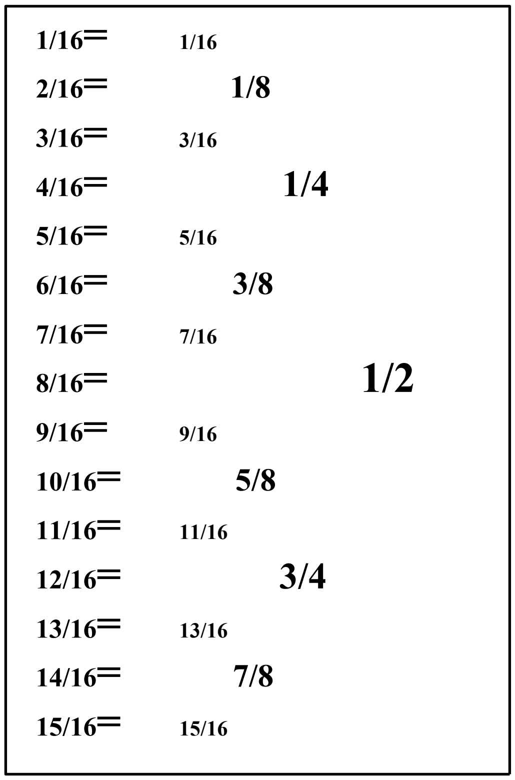
c. **def** `print_ruler(n)`

This function is given a positive integer `n`, and prints a vertical ruler of $2^n - 1$ lines. Each line contains '-' marks as follows:

- The line in the middle ($\frac{1}{2}$) of the ruler contains *n* '-' marks
- The lines at the middle of each half ($\frac{1}{4}$ and $\frac{3}{4}$) of the ruler contains (*n*-1) '-' marks
- The lines at the $\frac{1}{8}, \frac{3}{8}, \frac{5}{8}$ and $\frac{7}{8}$ of the ruler contains (*n*-2) '-' marks
- And so on …
- The lines at the $\frac{1}{2^k}, \frac{3}{2^k}, \frac{5}{2^k}, \ldots, \frac{2^k-1}{2^k}$ of the ruler contains *1* '-' mark

For example, `print_ruler(4)`, should print (only the blue marks):

```
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
```



Hints:

1. Take for n=4: when finding `print_ruler(4)`, try to think first **what** `print_ruler(3)` does, and how you can use it to print a ruler of size 4. Then, generally identify what `print_ruler(n-1)` is **supposed** to print, and use that in order to define how to print the ruler of size n.
2. You may want to have more than one recursive call
3. It looks much scarier than it actually is

Question 4:

Give a **recursive** implement to the following function:

```
def list_min(lst, low, high)
```

The function is given `lst`, a list of integers, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should find and return the minimum value out of all the elements at the position *low, low+1, ..., high* in `lst`.

<u>Question 5:</u>

Give a **recursive** implement to the following functions:

a) `def count_lowercase(s, low, high):`

The function is given a string `s`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.
The function should return the number of lowercase letters at the positions *low, low+1, …, high* in `s`.

b) `def is_number_of_lowercase_even(s, low, high):`

The function is given a string `s`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.
The function should return `True` if there are even number of lowercase letters at the positions *low, low+1, …, high* in `s`, or `False` otherwise.

<u>Question 6:</u>

Give a **recursive** implement to the following function:

`def appearances(s, low, high)`

The function is given a string `s`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.
The function should return a dictionary that stores a mapping of characters to the number of times they each appear in `s`. That is, the keys of the dictionary should be the different characters in `s`, and their associated values should be the number of times each of them appears in `s`.

For example, the call `appearances(”Hello world”, 0, 10)` could return:
`{’e’:1, ’o’:2, ’H’:1, ’l’:3, ’r’:1, ’ ’:1, ’d’:1,’w’:1}`.

<u>Note</u>: A dictionary is a mutable object. Use that property to **update** the dictionary, returned from your recursive call.

<u>Question 7:</u>

Give a **recursive** implement to the following function:

`def split_by_sign(lst, low, high)`

The function is given a list `lst` of non-zero integers, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.
The function should reorder the elements in `lst`, so that all the negative numbers would come before all the positive numbers.

<u>Note</u>: The order in which the negative elements are at the end, and the order in which the positive are at the end, doesn’t matter, as long as all he negative are before all the positive.

__Question 8:__

A ***nested list of integers*** is a list that stores integers in some hierarchy. That is, its elements are integers and/or other nested lists of integers.
For example `nested_lst=[[1, 2], 3, [4, [5, 6, [7], 8]]]` is a nested list of integers.

Give a **recursive** implement to the following function:

```
def flat_list(nested_lst, low, high)
```

The function is given a nested list of integers `nested_list`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.
The function should flatten the sub-list at the positions *low, low+1, ..., high* of `nested_list`, and return this flattened list. That is, the function should create a new 1-level (non hierarchical) list that contains all the integers from the *low…high* range in the input list.
For example, when calling `flat_list` to flatten the list `nested_lst` demonstrated above (the initial call passes `low=0` and `high=2`), it should create and return `[1, 2, 3, 4, 5, 6, 7, 8]`.

__(Extra Credit) Question 9:__

Given an ordered list *L*. A ***permutation*** of *L* is a rearrangement of its elements in some order. For example *(1, 3, 2)* and *(3, 2, 1)* are two different permutations of *L=(1, 2, 3)*.

Implement the following function:

```
def permutations(lst, low, high)
```

The function is given a list `lst` of integers, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.
The function should return a list containing all the different permutations of the elements in `lst`. Each such permutation should be represented as a list.

For example, if `lst=[1, 2, 3]`, the call `permutations(lst, 0, 2)` could return `[[1, 2, 3], [2, 1, 3], [1, 3, 2], [3, 2, 1], [3, 1, 2], [2, 3, 1]]`

<u>Hint</u>: Think recursion!!!
Take for example `lst=[1, 2, 3, 4]`: to compute the permutation list of `lst`, try to first write down the list containing all permutations of `[1, 2, 3]`, then, think how to modify it, so you get the permutation list of `[1, 2, 3, 4]`.