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 8

Recursion

Date Due: Thursday, November 5, 2020 **Total Marks: 24**

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.
- 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. **Put your name, NSID, student number and instructor's name 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, zip documents, even if you think it will help.
- **Programs must be written in Python 3.** Marks will be deducted with severity if you submit code for Python 2.
- **Assignments must be submitted to Moodle.** There is a link on the course webpage that shows you how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Questions are annotated use descriptors like "easy" "moderate" and "tricky". All students should be able to obtain perfect grades on "easy" problems. Most students should obtain perfect grades on "moderate" problems. The problems marked "tricky" may require significantly more time, and only the top students should expect to get perfect grades on these. We use these annotations to help students be aware of the differences, and also to help students allocate their time wisely. Partial credit will be given as appropriate, so hand in all attempts.

Question 1 (8 points):

Purpose: To practice recursion

Degree of Difficulty: Moderate.

At each of a number of archaeological sites many cylindical blocks have been found that appear to be part of ancient columns that have toppled. For each site your task is to determine if the blocks could be stacked to form a column of a certain height. All of the blocks have a height, which we will call their size, which is an integer number of cubits. Two blocks may have the same size. For each site you have a `list` of the sizes of the blocks found at that site.

Your task is to write a **recursive function** that determines whether or not it is possible to form a column of a given height using only the blocks whose sizes are given in list. Each block can be used at most once in building a column.

Your function should have 2 parameters: `height` which is an integer giving the desired height of a column in cubits, and `block_sizes` which is a list of the sizes of the available blocks at a site, these sizes are integer number of cubits. Your function should return True is it is possible to build a column of the desired height, and False if it is not possible.

When thinking about forming a column from a list of blocks you should begin by considering the first block in the list. Should this first block be part of the column or not? There is no way to know in advance, thus you should consider both possibilities.

If you don't include the first block, then you need to try to form the column with the other blocks. If you do use this first block, then with the other blocks you need to form a column that is shorter than the desired height by the size of the first block. Note that it is always possible to form a column of height zero by using zero blocks.

Test your function using the following block size information from sites 0 and 1.

```
site0 =[2,5,5,6]
site1 = [3,3,5,7,7,9]
```

For each site print a list of all column sizes from 1 to 50 that can be formed from the discovered blocks.

Sample Run

Form site 0 you should be able to get output that looks like this:

```
The heights of columns that can be built from blocks found at site 0 are
[2, 5, 6, 7, 8, 10, 11, 12, 13, 16, 18]
```

What to Hand In

- A document entitled `a8q1.py` containing your finished program, as described above.

Be sure to include your name, NSID, student number and instructor's name at the top of all files.

Evaluation

- 1 mark: The recursive function contains a recursive call.
- 1 mark: The base cases are correct.
- 4 marks: The recursive case is correct such that the recursive function returns the correct values.
- 1 mark: The doc-strings are sufficiently descriptive, indicating the purpose and requirements of each function.
- 1 mark: The main program correctly prints column heights between 1 and 50 that can be constructed by calling the recursive function
- **-3 marks: Global variables and/or loops are used in the recursive function.**
- **-1 mark: for missing name, NSID and student number at top of file**

Question 2 (8 points):

Purpose: To practice recursion

Degree of Difficulty: Moderate.

In an old house in Russia several chests of full of hollow dolls have been discovered. These dolls appear to be from sets of matryoshka dolls. A set of matryoshka dolls is a series of dolls where doll1 fits inside doll2, doll2 fits inside doll3, and so on. Your task is to determine the sizes of the largest sets of matryoshka dolls that can be formed from each chest. Inside the chests the dolls have been all jumbled up. Fortunately, each doll has a unique name and your assistant has gone throught the dolls in each chest and determined which dolls fit inside which others. This has not been easy, as one doll may be shorter than another, but also may have larger feet.

For this problem, you will be given information about each chest as a `dictionary`. The `keys` to this dictionary are doll names, and the `value` for each key is a list of other dolls in the chest that fit inside it. For a small doll, where no other dolls fit inside it, there will be no `dictionary` entry with the small dolls name as the key. If there is a `dictionary` entry with a certain dolls name then there is at least one doll that fits inside it (ie there are no empty list as values in the `dictionary`)

The following is a sample of what this dictionary might look like:

```
chest1 = {
    "Dafna" : ["Vera", "Lada", "Alla","Zoe"],
    "Yana" : ["Zoe"],
    "Vera" : ["Inga"],
    "Alla" : ["Zoe"]
}
```

In the example above, the doll **Dafna** can contain **Zoe**, but not **Yana**.

Your first task is to write a **recursive function** that will determines the maximum number in a set of matyoshka dolls whose outermost doll's name is given. Your function should have 2 parameters: the `chest` which is the python `dictionary` containing the information about doll containment in the indicated chest, and a `doll's name` . Your function should return the maximum number of dolls in a set of matyoshka dolls, that can be formed from dolls in the given chest, whose outermost doll has the given name. If the given outermost doll contains no others in the chest then one should be returned.

Your second task to write a **function** that prints the maximum number in a set of matyoshka dolls, that includes at least two dolls, for each outermost doll in the given chest. This function has a single parameter: the `chest` which is the python `dictionary` containing the information about doll containment in the indicated chest. This second function will not be recursive but will call your first recursive function.

Finally, in the main program you will call your second function for each of the two provided chest dictionaries.

Provided File

We are providing you with 2 chest dictionaries to use; you can find these on Moodle in `asn8q2-starter.py`. You can just copy/paste the dictionary literals into your code, or rename the file to a8q2.py and put your own code within.

Sample Run

If you call your second function with the provided `chest1`, you should be able to get output that looks like this:

```
From the chest
The maximum size of a set Matyoshka Dolls with outermost doll Dafna is 3
The maximum size of a set Matyoshka Dolls with outermost doll Yana is 2
The maximum size of a set Matyoshka Dolls with outermost doll Alla is 2
The maximum size of a set Matyoshka Dolls with outermost doll Vera is 2
```

Note that the order that the dolls are considered will vary in each run.

What to Hand In

- A document entitled `a8q2.py` containing your finished program, as described above.

Be sure to include your name, NSID, student number and instructor's name at the top of all files.

Evaluation

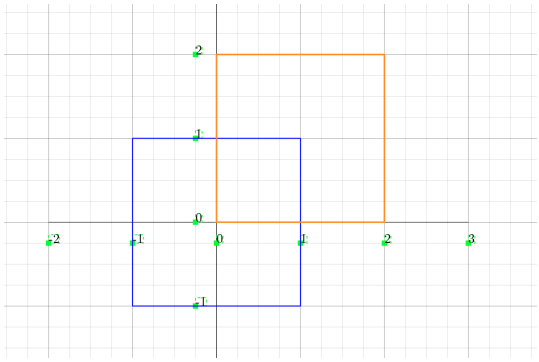
- 1 mark: The recursive function is called correctly using appropriate arguments
- 1 mark: The base case is correct.
- 4 marks: The recursive case is correct such that the recursive function returns the correct values.
- 1 mark: The doc-strings are sufficiently descriptive, indicating the purpose and requirements of each function.
- 1 mark: The second function correctly prints the maximum number in a set of matyoshka dolls, that includes at least two dolls, for each outermost doll in the given chest
- **-3 marks: Global variables and/or loops are used in the recursive function.**
- **-1 mark: for missing name, NSID and student number at top of file**

Question 3 (8 points):

Purpose: To practice recursion

Degree of Difficulty: Moderate.

For this problem you need to determine which labels can be drawn on a map without having any two of them overlap. Each potential label is a square with side length 2 whose lower left corner has integer coordinates. For example in the figure the blue and orange squares represent potential labels for a map, but since they overlap only one of them can be drawn. The blue square is given by its lower left corner coordinates (-1,-1), and the orange square by (0,0).



Your task is to write a **recursive function** that determines which squares (potential labels), from a given list of squares, should be added to another list of nonintersecting squares (confirmed labels) so as to display the maximum number of labels. The function should return a list of labels that includes the intially confirmed labels plus as many as possible of the potential labels.

Your function should have 2 parameters: The first is a list of 2 by 2 squares that are potential labels on a map. Each square's lower left corner coordinates are given as a tuple of two integers. The second is a list of 2 by 2 nonintersecting squares (confirmed labels) Again, each square's lower left corner coordinates are given as a tuple of two integers.

Approach

When thinking about adding to the list of confirmed labels you should begin by considering the first square in the list of potential labels. Should this first square be added? There is no way to know in advance, thus you should consider both possibilities to find a labeling with the most square labels.

If you don't include the first square, then you need to try to create the largest labeling by adding to the confirmed labels using the rest of the potential labels. Note that you can only add the first square to the list of confirmed labels if it does not intersect with any of them. If you do add this first square, then to find the largest possible labeling you still need to consider the other potential labels.

Provided Functions

We are providing two functions for you to use. These are contained in the file `a8q3_starter.py` available on the moodle site. You may copy and paste these into your program, or rename the file to `a8q3.py` and then add your solution code.

```
def intersection_2sq(s1,s2):
    """
    This function determines whether or not the two given squares intersect. The two
    squares are given by the integer coordinates of their lower left corner.
    Both squares have side length 2.
    Parameters: s1,s2: Each is a 2 by 2 square whose lower left corner coordinates
    is given as a tuple of two integers
    Returns Boolean: True if the two squares intersect, and False if they do not.
    """
      return s1[0]-1 <= s2[0] and s2[0] <= s1[0] + 1 and \
               s1[1] - 1 <= s2[1] and s2[1] <= s1[1] + 1


def intersection_inlist(test_square,List):
    """
    This function determines if a given test_square intersects with any of the squares
    in the given list of squares. All of the squares have side length 2.
    Parameter1 test_square:A 2 by 2 square whose lower left corner coordinates is
    given as a tuple of two integers
    Parameter2 List: A list of 2 by 2 squares whose lower left corner coordinates
    are given as a tuple of two integers
    Returns Boolean:  True is the test_square intersects with any of the squares
    in the given list of squares,
    False if there is no intersection.
    """
    for sq in List:
        if intersection_2sq(test_square,sq):
            return True
    return False
```

Testing

Test your function using the following three sets of potential labels. When you call your recursive function from the main program you will need to pass it one of these lists of potential labels, and also an empty list (confirmed labels).

```
squarelist1 = [(-1,-1),(0,0),(1,1)]
squarelist2 = [(2,3),(6,8),(6,9),(9,1),(3,4),(2,4)]
squarelist3 = [(-3,0),(-2,1),(-1,2),(0,1),(1,0),(0,-1),(-1,-2),(-2,-1)]
```

Sample Run

Using `squarelist1 = [(-1,-1),(0,0),(1,1)]` as the list of potential labels you should be able to get output that looks like this:

```
From among the labels [(-1, -1), (0, 0), (1, 1)]
The following can be drawn without intersection [(-1, -1), (1, 1)]
```

What to Hand In

- A document entitled `a8q3.py` containing your finished program, as described above.

Be sure to include your name, NSID, student number and instructor's name at the top of all files.

Evaluation

- 1 mark: The recursive function contains a recursive call.
- 1 mark: The base case is correct.
- 2 marks: The recursive logic is correct when the first square from the potential list IS added.
- 2 marks: The recursive logic is correct when the first square from the potential list is NOT added.
- 1 mark: The doc-strings are sufficiently descriptive, indicating the purpose and requirements of each function.
- 1 mark: The main program correctly prints the best labeling for each of the three input lists by calling the recursive function
- **-3 marks: Global variables and/or loops are used in the recursive function.**
- **-1 mark: for missing name, NSID and student number at top of file**