COSC 130 – HW 05 Instructions

General Instructions

Create a new notebook named **`HW_05_YourLastName.ipynb`** and complete problems 1 – 8 described below. Download the files **`diamonds_partial.txt`** and **`titanic_partial.txt`**, placing them in the same folder as your notebook.

Any set of instructions you see in this document with an orange bar to the left will indicate a place where you should create a markdown cell. For each new problem, create a markdown cell that indicates the title of that problem as a level 2 header.

Any set of instructions you see with a blue bar to the left will provide instructions for creating a single code cell.

Read the instructions for each problem carefully. Each problem is worth 6 points. An additional 2 points are allocated for formatting and following general instructions.

The assignment should be completed using only base Python commands. Do not use any external packages, except for **`random`** (see below).

Note on Problems 4 – 8

Problems 4 – 8 are intended to provide you with experience writing recursive functions. Iterative techniques (using loops) exist for solving each of these problems, but no points will be awarded for solutions that do not involve recursion. The solutions for Problems 5 and 8 will likely make use of loops along with recursion, but not loops are needed (or should be used) in Problems 4, 6, and 7.

Assignment Header

Create a markdown cell with a level 1 header that reads: "COSC 130 - Homework 05". Add your name as a level 3 header

We will use randomly generated lists to test some of the functions created in this notebook. To generate such lists, we will need to use the **`random`** package.

Add a code cell containing the following import statement: **`import random`**

(The rest of this page has been deliberately left blank. Problem 1 is described on the next page.)

Problem 1: Argument Sort

In this problem, you will write a function named **`argsort()`** that sorts a list, but rather than returning a list of elements in the sorted version of the list, it will return a list of indices indicating the positions of the sorted values in the original list.

The function should accept two parameters named **`x`** and **`reverse`**.

- **`x`** is expected to be a list of elements of the same data type.
- **`reverse`** is expected to be a Boolean value. It should have a default value of **`False`**.

Let's take a careful look at what this function is supposed to accomplish.

Assume that **x** is a list given as follows: **`x = ['b', 'c', 'a', 'e', 'd']`**. If we were to sort this list, we would get **`['a', 'b', 'c', 'd', 'e']`**. The function **`argsort(x)`** should return the list **`[2, 0, 1, 4, 3]`**.

To understand how the return value of **`argsort()`** is constructed, note the following:

- The 1st element of the sorted list is **`'a'`**, which is at index 2 of **x**. So the 1st element of the returned list is 2.
- The 2nd element of the sorted list is **`'b'`**, which is at index 0 of **x**. So the 2nd element of the returned list is 0.
- The 3rd element of the sorted list is **`'c'`**, which is at index 1 of **x**. So the 3rd element of the returned list is 1.
- The 4th element of the sorted list is **`'d'`**, which is at index 4 of **x**. So the 4th element of the returned list is 4.
- The 5th element of the sorted list is **`'e'`**, which is at index 3 of **x**. So the 5th element of the returned list is 3.

There are multiple ways to perform **`argsort()`**. Perhaps the simplest method makes use of tuples in a clever, but not obvious way. The process works like this:

1. We use **x** to create a new list containing tuples. The first element of each tuple is an element of **x**, while the second element is the index corresponding to that element. For the list **x** provided above, this step would result in the following list: **`[('b', 0), ('c', 1), ('a', 2), ('e', 3), ('d', 4)]`**
2. We then sort the list of tuples. When doing so, the tuples will be sorted according to their **first** element. This would give us: **`[('a', 2), ('b', 0), ('c', 1), ('d', 4), ('e', 3)]`**
3. We then loop over the sorted list of tuples, extracting the second element of each tuple. The resulting list will be our desired return value. In this case, we would get: **`[2, 0, 1, 4, 3]`**

This function will be used later as part of the Group Project.

Write a function named **`argsort()`** that takes two parameters named **`x`** and **`reverse`**, as described on the previous page. The parameter **`reverse`** should have a default value of **`False`**.

This function should perform the argsort process described above. If **reverse** is **True**, then the sorting should be performed in descending order and the resulting list of indices should be given in reverse order.

The steps in calculating the return value are as follows:

1. Create a list named **`tuple_list`** that consists of tuples of the form **`(x[i], i)`**, where **`i`** refers to the index of an element in **x**. This can be done with a loop or a list comprehension.
2. Use **`sorted()`** to sort **`tuple_list`**, storing the result in a list named **`sorted_tuples`**. Use the **`reverse`** parameter to determine the direction of the sorting.
3. Create a list named **`idx_list`** by extracting the second element of each tuple in **`sorted_list`**.
4. Return **`idx_list`**.

We will now test the **`argsort()`** function on a list of integers.

In a new code cell, create the following list: **`list1 = [76, 81, 30, 47, 50, 18, 23, 49]`**
Call **`argsort()`** on **`list1`**, sorting in ascending order. Print the result.

We will now test the **`argsort()`** function on a list of strings.

In a new code cell, create the following list: **`list2 = ['Chad', 'Beth', 'Emma', 'Alex', 'Drew', 'Fred']`**
Call **`argsort()`** on **`list2`**, sorting in descending order. Print the result.

Problem 2: Process Lines of Text

In this problem, you will write a function named **`process_line()`** that accepts a string representing a sequence of distinct values. The function will split the string into individual strings, coerce each string into the desired data type, and will then return the results as a list.

The function should accept three parameters named **`line`**, **`schema`**, and **`sep`**.

- **`line`** should contain a string that is to be split into separate values.
- **`schema`** should be a list of data types indicating the desired types for each of the values in **`line`**.
- **`sep`** should be a string indicating the character used to separate the different values in **`line`**.

As an example, consider the following string:

```
test_line = '6,174,Blah,Hello World,7.37'
```

We wish to separate this into smaller strings by splitting it at commas. We will then coerce each resulting token into the appropriate data type according to the following schema:

```
test_schema = [int, int, str, str, float]
```

The function call **`process_line(test_line, test_schema, ',')`** should return the following list:

```
[6, 174, 'Blah', 'Hello World', 7.37]
```

Write a function named **`process_line()`** that accepts three parameters **`line`**, **`schema`**, and **`sep`** as described above.

The function should perform the following steps:

1. Use the **`split()`** method to split the string **`line`** on the character provided by **`sep`**. Store the resulting list in a variable named **`tokens`**.
2. Create an empty list named **`result`**.
3. Simultaneously loop over the elements of the lists **`tokens`** and **`schema`** (which are intended to have the same number of elements). Each time the loop executes, perform the following steps:
   a. Store the current element of **`tokens`** in a variable named **`t`**.
   b. Store the current element of **`schema`** in a variable named **`dt`**.
   c. Note that **`dt`** will contains a data type, while **`t`** will hold a string. Coerce **`t`** into a value with the desired data type using **`dt(t)`**. Append the converted value into the list **`result`**.
4. Return **`result`**.

We will apply this function in the next example, but we will first test it to make sure that it is working correctly.

In a new code cell, create the following objects:

```
test_schema = [int, int, str, str, float]
test_line = '6,174,Blah,Hello World,7.37'
```

Use the **`process_line()`** function to split the string **`test_line`** at the commas, coercing the individual tokens into the data types stored in **`test_schema`**. Print the result.

Problem 3: Processing File Input

In this problem, you will write a function named **`read_file_to_list()`** that will read rows of text from a data file, tokenize the rows into individual values, coerce those values into the proper data types, and then return the results in the form of a list of lists.

The function should accept three parameters named **`path`**, **`schema`**, and **`sep`**.

- **`path`** should be a string representing the path to a data file.
- **`schema`** should be a list of data types indicating the desired types for the columns stored in the data file.
- **`sep`** should be a string indicating the character used to separate values stored in the lines of the data file.

We will assume that the first line of the data files used will contain header information that assigns a name to each column. This line will be tokenized (split), but the values will be left as strings, rather than being coerced according to **`schema`**.

As an example, assume that a datafile located at **`path`** contains the following lines of text:

```
Name,Age,PayRate
Anna,27,15.25
Bradley,31,16.75
Catherine,23,15.50
```

Define **`my_schema = [str, int, float]`**. Then the function call **`read_file_to_list(path, my_schema, ',')`** should return the following list of lists:

```
[['Name', 'Age', 'PayRate'], ['Anna', 27, 15.25], ['Bradley', 31, 16.75], ['Catherine', 23, 15.5]]
```

Write a function named **read_file_to_list()** that accepts three parameters **`path`**, **`schema`**, and **`sep`** as described above.

The function should perform the following steps:

1. Use **`with`**, **`open()`**, and **`read()`** to read the contents of the file into a string named **`contents`**.
2. Use **`split()`** to separate the string into a list named **`lines`** by splitting on the newline character.
3. Create an empty list named **`data`**. This will eventually contain the list that is to be returned.
4. The first line contains header information and will be processed differently from the other lines. Split the first string in **`lines`** on **`sep`**. Store the resulting list into the list **`data`**.
5. We no longer need the first element of **`lines`**. For convenience, you may delete it.
6. Loop over the remaining elements of **`lines`**. Each time the loop executes, use the function **`process_lines()`** to process the current line, appending the resulting list into **`data`**. Use the values provided to the parameters **`schema`** and **`sep`**.
7. Return **`data`**.

We will now test the **`read_file_to_list()`** function on two small data files. We will start with a data file that contains 10 observations from the Diamonds Dataset. Make sure that the file **`diamonds_partial.txt`** is in the same directory as your notebook. I suggest opening this file so that you can see what its contents look like.

Use **`read_file_to_list()`** to read the contends of the file **`diamonds_partial.txt`**. Individuals values within the rows of this data file are separated by commas. The datatypes for the columns in this data set are given by the following schema: **`[float, str, str, str, int]`**. Store the resulting list in a variable named **`diamond_data`**. Use a loop to print the lists contained in **`diamond_data`** with each list appearing on its own line.

We will now test our function with a file containing 10 observations from the **Titanic Dataset**. Make sure that the file **`titanic_partial.txt`** is in the same directory as your notebook. I suggest taking a look at the contents of this file.

Use **`read_file_to_list()`** to read the contends of the file **`titanic_partial.txt`**. Individual values within the rows of this data file are separated by tab characters. The datatypes for the columns in this data set are given by the following schema: **`[int, int, str, str, int, float]`**. Store the resulting list in a variable named **`titanic_data`**. Use a loop to print the lists contained in **`titanic_data`** with each list appearing on its own line.

Note that this function will be used again later in the course as part of the Group Project.

Problem 4: Recursive Product

In this problem, you will write a function named **`recursive_product()`** that recursively calculates the product of the elements in a list. The function should accept a single parameter **`x`**, which is intended to be a list, and should return the product of elements in that list.

Write a function **`recursive_product()`** that accepts a single parameter **`x`**.

The function should calculate and return the product of the elements in **`x`** by performing the following steps:

1. If the length of **`x`** is equal to 1, then return the single value stored in **`x`** (and not the list **`x`** itself).
2. Otherwise, pass a list consisting of every element of **`x`** EXCEPT the first element to **`recursive_product()`**, storing the result in a variable named **`temp`**.
3. Return the product of **`temp`** and the first element of **`x`**.

We will now test the function.

In a new code cell, create the following list: **`factors = [11, 8, 17, 9, 18, 10]`**. Pass the list **`factors`** to **`recursive_product()`** and print the value returned.

Problem 5: Greatest Common Divisor

In this problem, you will use recursion to write a function named **`gcd()`** that will accept two parameters **a** and **b**, which are assumed to be integers. The function will return the greatest common divisor of **a** and **b**. In other words, it will return the largest number that evenly divides both integers.

Write a function **`gcd()`** that accepts two parameters **a** and **b**.

The function should calculate and return the greatest common divisor of **a** and **b** by performing the following steps:

4. If **b** is equal to 0, then return **a**.
5. If **b** is not equal to 0, then return **`gcd(b, r)`**, where **r** is the remainder that results from dividing **a** by **b**.

We will now test the function.

In a new code cell, call **`gcd()`** on each of the following pairs of integers. Print the result of each function call.

- 72 and 300
- 180 and 210
- 20 and 400
- 30 and 77

Problem 6: Flattening Nested Lists

In this problem, you will use recursion to write a function named **`flatten()`** that accepts a single parameter **`x`**, which is expected to be a list. Some of the elements of **`x`** might themselves be lists, and some of the elements of those lists could again be lists, and so on. We will assume that the lists could be nested within each other to an arbitrary depth. The function should "flatten" the structure of the nested lists by creating a single list that contains all of the non-list values stored within the structure, but no other lists.

For example, let **`nested_list = [1, [8, [5, 6, [4, 9]]], [7, 2, 3]]`**. Then **`flatten(nested_list)`** should return the list **`[1, 8, 5, 6, 4, 9, 7, 2, 3]`**.

Write a function **`flatten()`** that accepts accepts a parameter **`x`** intended to represent a list that perhaps contains other lists. The function should return a flattened version of **`x`** by performing the following steps:

1. Create an empty list named **`flat_list`**.
2. Loop over the elements of **`x`**. For each iteration of the loop, perform the following steps:
    a. If the current element of **`x`** is a list, then called **`flatten()`** on that element, concatenating the result to **`flat_list`**.
    b. If the current element of **`x`** is not a list, then append it to **`flat_list`**.
3. Return **`flat_list`**.

We will now test the function.

In a new code cell, call **`flatten()`** on each of the following lists. Print the result of each function call.

- `[1, 2, 3]`
- `[1, [2], [3]]`
- `[1, [2], [[4, 5], 6]]`
- `[1, [8, [5, 6, [4, 9]]], [7, 2, 3]]`

Problem 7: Binary Search

In this problem, you will write a function named **`binary_search()`** with two parameters **`x`** and **`item`**. The parameter **`x`** should be a list containing elements with a single data type, and that is assumed to have been sorted in increasing order. The parameter **item** should contain a value of the type that is contained in **x**.

The function will apply a divide-and-conquer technique to recursively search the list **`x`** for an occurrence of the value stored in **`item`**. The function should return **`True`** if **`item`** is found in **`x`**, and should return **`False`** otherwise.

Write a function **`binary_search()`** that accepts two parameters named **`x`** and **`item`**.

The function should return a boolean value indicating if **`item`** was found in **`x`** by performing the following steps:

1. First check if **`item`** is less than the first element of **`x`** or greater than the last element of **`x`**. Since **`x`** is assumed to be sorted, if either of these conditions are true, then **`item`** is not in **`x`** and you can return **`False`**.
2. We split the list **`x`** into two pieces by selecting an element in the middle of the list. Create a variable named **`mid`** that is equal to the length of **`x`** divided by 2, rounded down. Use **`int()`** to coerce the result into an integer.
3. If **`item`** is equal to **`x[mid]`**, return **`True`**.
4. If **`item`** is less than **`x[mid]`**, then **`item`** would have to be in the first half of **`x`** (if it is there at all). Slice out the elements of **`x`** up to the index **`mid`**, pass this sub-list and **`item`** to **`binary_search()`**, and return the result.
5. If **`item`** is greater than **`x[mid]`**, then **`item`** would have to be in the last half **`x`** (if it is there at all). Slice out the elements of **`x`** after the index **`mid`**, pass this sub-list and **`item`** to **`binary_search()`**, and return the result.

We will now test this function.

In a new code cell, create a randomly generated list using the following lines of code:

```
random.seed(1)
random_list_1 = sorted(random.choices(range(100), k=100))
```

Use a loop to search **`random_list_1`** for each integer 0 – 9. Print the results of each search in the following format:

```
N - XXXX
```

The character **N** should be replaced with the integer being search for and the characters **XXXX** should be replaced with either **`True`** or **`False`**.

Problem 8: Quicksort

We know that Python provides the function **`sorted()`** which can be used to sort a list. But we have not seen how this function works "underneath the hood". There are many possible sorting algorithms that can be used to sort a list. In this problem we will explore a recursive sorting algorithm named Quicksort.

Your solution in this problem should NOT make use of the functions `sort()` or `sorted()`. In fact, the only built-in functions or methods it should use are `len()` and `append()`.

In this problem, you will write a function named **`quicksort()`** with one parameter named **`x`**, which is assumed to be a list of values of a single data type. The function should create and return a sorted copy of **`x`**. The function will select an element of **`x`** to act as a "pivot" upon which the list will be split. All elements of **`x`** less than the pivot will go into a list named **`low`** and all elements greater than or equal to the pivot will go into a list named **`high`**. The **`quicksort()`** function will then be recursively applied to the lists **`low`** and **`high`**.

Write a function **`quicksort()`** that accepts a single parameter named **`x`**.

The function should return a sorted copy of **`x`** by performing the following steps:

1. First check to see if the length of **`x`** is less than or equal to 1. If so, return **`x`**.
2. Set **`pivot`** equal to the first element of **`x`**.
3. Store **`pivot`** in a single-element list named **`mid`** and create empty lists named **`low`** and **`high`**.
4. Loop over all elements of **`x`** EXCEPT for the first element (which is the pivot). For each such element, if that element is less than the pivot, then append it to **`low`**, otherwise append it to **`high`**.
5. Call **`quicksort()`** on **`low`**, storing the result in a variable named **`sorted_low`**.
6. Call **`quicksort()`** on **`high`**, storing the result in a variable named **`sorted_high`**.
7. Concatenate **`sorted_low`**, **`mid`**, and **`sorted_high`** into a single list, and then return the result.

We will now test this function on a randomly generated list.

In a new code cell, create a randomly generated list using the following line of code:

```
random.seed(2)
random_list_2 = sorted(random.choices(range(100), k=20))
```

Print **`random_list_2`**. Then call **`quicksort()`** on **`random_list_2`** and print the list returned by that function.

We will test the function on a second randomly generated list.

In a new code cell, create a randomly generated list using the following line of code:

```
random.seed(3)
random_list_3 = sorted(random.choices(range(100), k=20))
```

Print **`random_list_3`**. Then call **`quicksort()`** on **`random_list_3`** and print the list returned by that function.

Submission Instructions

When you are done, click **Kernel > Restart and Run All**. If any cell produces an error, then manually run every cell after that one, in order. Save your notebook, and then export the notebook as an HTML file. Upload the HTML file to Canvas and upload the IPYNB file to CoCalc, placing it in the **Homework/HW 05** folder.