22C:16 CS:1210 Final Exam

May 15th, 10 am to noon

Instructions:

- This is an open notes exam and you have 2 hours to complete it. There are 4 problems in the exam and these appear on 9 pages.
- Make sure that you do not have any electronic devices (laptops, phones, calculators, dictionaries, etc.) on your desk; you are not allowed to access these during your exam.
- Write as neatly as you can.
- Show all your work, especially if you want to receive partial credit.

Name:

Circle your discussion section:

SCA (M, W evening) A01 (9:30-10:20) A02 (11-11:50) A03 (12:30-1:20)

A04 (2:00-2:50) A06 (3:30-4:20)

---

1. (**30 points**) On *Lists and Strings.*

   (a) (**10 points**) Suppose that `L = [(4.4, "Rocky"), (4.9, "Fargo"), (3.5, "Congo"), (5.1, "Argo"), (3, "Jaws"), (5, "Hamlet")]`. Evaluate each expression and write down its *value*.

   (a) `[x for y in L for x in y]`

   (b) `[(x[1], x[0]) for x in sorted(L, reverse=True)]`

   (c) `dict(L)`

   (d) `[x[0] for x in L if "go" in x[1]]`

(e) `L[2:5:2]`

(b) (**20 points**) Your task is to write a function called `evaluateExpression` that evaluates a given expression. Expressions you might be given are made up of variable names and non-negative integer constants separated by the addition operator. An example of an expression you might be given is `" sum +10 + 2500 + i"`. Note that the given expression is represented as a string. It is also worth noting that zero or more blanks could appear between any consecutive pair of objects in an expression. In addition, your function is also given a list of variable names and a parallel list of associated values. For example, to evaluate the expression above, your function needs to know the value of `sum` and `i`. If your function is given two additional lists, e.g., `["sum", "i"]` and `[10, 2]`, then it could interpret this to mean that the value of `sum` is 10 and the value of `i` is 2. Thus the expression would evaluate to 2522. Thus the function call

```
evaluateExpression(" sum + 10 + 2500 + i", ["sum", "i"], [10, 2])
```

would return 2522. Using this description, write a function with the following header:

```
evaluateExpression(expr, varList, valueList)
```

(**Hint:** Our solution has 8 lines of code and it uses string methods `split`, `strip`, and `isalpha` and the list method `index`.)

2. (**40 points**) On *dictionaries*.

(a) (**10 points**) Suppose that `D` is the dictionary {`12:1, 15:2, 2:17, 1:8, 8:17, 17:1`}. Write down the value of `D` after each of the following Python statements. Evaluate each statement starting with the same value of the dictionary `D`, shown above.

(a) `del D[2]`

(b) `D[D[17]] = D[2]`

(c) `D.update({1:17, 2:1})`

(d) `D.update({15:15})`

(e) `D.clear()`

(b) (**15 points**) Let `D` be the dictionary that represents the *word network* that we constructed in implementing a solution to the word ladder game problem. Thus every valid 5-letter word is a key in `D` and for any valid 5-letter word `w`, `D[w]` is a list consisting of all valid 5-letter words that can be obtained from `w` by changing exactly one letter in `w`. In other words, the value `D[w]` of the key `w` is the list of all words that can be reached in 1 hop from `w` in the word network. For example, `D["hello"]` is the list `["cello, "hallo", "hells", "hullo", "jello"]`.

Given below is a partial implementation of a function called `twoHopNeighbors` with function header:

```
def twoHopNeighbors(w, D):
```

that returns the list of valid 5-letter words that can be reached from w in *exactly two hops, but no fewer*. For example, a call to `twoHopNeighbors("hello", D)` should return

```
["cells", "halls", "bells", "dells", "fells", "heals",
"heels", "helms", "helps", "hills", "hulls", "jells", "sells",
"tells", "wells", "yells", "jelly"]
```

Note that a word such as `"hallo"` is not in this list because even though it can be reached in two hops (`"hello"` to `"hullo"` to `"hallo"`), it can also be reached in one hop. Here is partially completed code for `twoHopNeighbors`. Your task is to fill in the blanks.

```
def twoHopNeighbors(w, D):
    twoHopsList = [] # variable to maintain list of 2-hop neighbors

    # Visit each neighbor of w
    for neighbor in D[w]:
        # Visit each neighbor of each neighbor of w
        # We need another for-loop for this

        for _____________________________________________:

            # Check if currently considered word should be added to twoHopsList
            # and add it if necessary

            if ____________________________________________________________:

                twoHopsList.append(_________________________)

    return twoHopsList
```

(c) (**15 points**) Let `D` be a dictionary that represents a list of student names along with their scores on some test. Assume that the scores are integers in the range 0 through 100 (inclusive of 0 and 100). Thus the keys in `D` are student names (strings) and the value associated with each key is an integer in the range 0 through 100. Thus, `D` might look like {`"Sam":8, "Jack":25, "Bob":28, "Duck":98, "Swift":100`}. Write a function `invert` with the following function header:

```
def invert(D):
```

The function `invert` returns a new dictionary whose keys are $0, 1, 2, \ldots, 10$. The value of a key k, $0 \leq k \leq 10$, is a list of student names (in any order) whose scores are between $10k$ and $10k + 9$ (inclusive). For example, the value of key 0 is the list of students with scores between 0 and 9 (inclusive), the value of key 1 is the list of students with scores between 10 and 19 (inclusive), etc. Calling `invert(D)` on `D =` {`"Sam":8, "Jack":25, "Bob":28, "Duck":98, "Swift":100`} yields {`0:["Sam"], 1:[], 2:["Bob", "Jack"], 3:[], 4:[], 5:[], 6:[], 7:[], 8:[], 9:["Duck"], 10:["Swift"]`}.

Using this description, implement the function `invert`.
(**Hint:** Our solution is 5 lines long.)

3. (**40 points**) On *recursion*

(a) (**15 points**) Given a length-n list `L` of *distinct* elements and an integer `k`, $1 \leq$ `k` $\leq n$, we want to write a function called `selection` that takes `L` and `k` as parameters and returns the `k`-th smallest element in `L`. For example, if `L = [5, 4, 1, 9, 10, 11, 2, 8]`, then `selection(L, 2)` returns 2 because 2 is the second-smallest element in `L`. Similarly, `selection(L, 5)` returns 8 because 8 is the fifth-smallest element in `L`.

A fast way of solving this problem is to use a "divide-and-conquer" approach similar to the one used in *quick sort*. As in the case of quick sort, we will first implement a more "general" `selection` function that finds the `k`-th smallest element in the slice `L[first:last+1]`. Thus we will implement a function called `generalSelection` with the following function header:

```
def generalSelection(L, k, first, last):
```

Here is a description of the algorithm we will use for this implementation.

(i) (**Divide** step) Call the `partition` function that we implemented for quick sort to obtain an index `p` such that `L[`i`]` $<$ `L[p]` for all i, where `first` $\leq i <$ `p` and `L[`i`]` $>$ `L[p]` for all i, where `p` $< i \leq$ `last`.

(iii) (**Conquer** step) Use the value of `p` and `first` to figure out the number of elements in `L[first:last+1]` that are smaller than `L[p]`. Comparing this quantity with `k` should tell us if `L[p]` is the `k`-th smallest element or whether we should be looking in the "left" slice `L[first:p]` or whether we should be looking in "right" slice `L[p+1:last+1]` for the element we are interested in.

Here is partially completed code for this problem. Your task is to complete it by filling in the four blanks.

```
def generalSelection(L, k, first, last):
    # Divide step
    p = partition(L, first, last)

    # The variable numSmaller is assigned the number of elements
    # in L[first:last+1] that are smaller than L[p]

    numSmaller = _________________________

    # Check if L[p] is the k-th smallest element
    if numSmaller + 1 == k:

        return ______________

    # Check if the k-th smallest element is in the "right" slice
    elif numSmaller+1 < k:

        return ___________________________________________________

    # Check if the k-th smallest element is in the "left" slice
    elif numSmaller+1 > k:

        return ___________________________________________________

# wrapper function
def selection(L, k):
    return generalSelection(L, k, 0, len(L)-1)
```

(b) (**10 points**) Here is the code for *merge sort*, discussed in class and posted on the course website. Note that We have inserted two print statements into the function `generalMergeSort`.

```
def merge(L, first, mid, last):
    i = first # index into the first half
    j = mid + 1 # index into the second half
    tempList = []

    while (i <= mid) and (j <= last):
        if L[i] <= L[j]:
            tempList.append(L[i])
            i += 1
        else:
            tempList.append(L[j])
            j += 1

    if i == mid + 1:
        tempList.extend(L[j:last+1])
    elif j == last+1:
        tempList.extend(L[i:mid+1])

    L[first:last+1] = tempList


def generalMergeSort(L, first, last):
    if first < last:
        mid = (first + last)/2
        generalMergeSort(L, first, mid)
        print L[first:mid+1]
        generalMergeSort(L, mid+1, last)
        print L[mid+1:last]
        merge(L, first, mid, last)
```

What output does the following function call produce:

`generalMergeSort(L, 3, 7)`

when `L = [10, 11, 2, 9, 1, 4, 8, 2, 12, 1, 15]`?

(c) (**15 points**) Write a *recursive* function to compute the maximum element of a given list. The algorithm you are required to use is this:

> the maximum element of a list `L` is the larger of element `L[0]` and the maximum of the `L[1:]`.

Use the following function header:

```
def maximum(L):
```

Make sure that you have taken care of all base cases.

4. (**40 points**) On *objects and classes*. We want to define a class called `employeeInfo` that a company uses to maintain a collection of employee records. Each employee record contains a *name*, a *social security number*, a *salary*, and an *employment start date*. In addition to an initialization method (`__init__`) and a representation method (`__repr__`), the class provides an `add` method for adding an employee record to the collection and a `remove` method for deleting an employee record. Here is an example of how a user might interact with the `employeeInfo` class:

```
>>> emp = employeeInfo()
>>> emp.add("Isaac Newton", 31415926, 1000000, "05152013")
>>> emp.add("Robert Boyle", 15793861, 5000000, "11152010")
>>> emp
Robert Boyle 15793861 5000000 11152010
Isaac Newton 31415926 1000000 05152013
>>> emp.add("Robert Hooke", 53589793, 200000, "04152012")
>>> emp
Robert Boyle 15793861 5000000 11152010
Isaac Newton 31415926 1000000 05152013
Robert Hooke 53589793 200000 04152012
>>> emp.remove(31415926)
>>> emp
Robert Boyle 15793861 5000000 11152010
Robert Hooke 53589793 200000 04152012
```

We assume that employees have distinct social security numbers and no employee appears twice in the collection of employee records. Therefore, we can use a dictionary-based implementation with the social security numbers acting as keys. Below we provide implementation of the initialization method and the `add` method:

```
class employeeInfo():

    def __init__(self):
        self.D = {}

    def add(self, name, ssn, salary, start):
        self.D[ssn] = [name, salary, start]
```

(a) (**10 points**) Implement the `remove` method.
(**Hint:** This just takes two lines of code including the function header.)

(b) (**15 points**) Implement the `__repr__` method. Recall that the `__repr__` method is required to return a string. The examples above of interacting with the `employeeInfo` class tell us that the string returned by `__repr__` contains information about each employee separated by the end-of-line character. Also, each employee's information contains the employee's name, employee's social security number, employee's salary, and employee's starting date, *in that order*, separated by a single blank character. Finally, note that the employee information appears in increasing order of social security numbers.

(c) (**15 points**) Now suppose that we change the implementation of the `employeeInfo` class, without changing how it behaves to an outside user. Specifically, instead of using a dictionary to store the collection of employee records, we will use a list. Below we provide part of the new implementation, namely the initialization method and the `__repr__` method:

```
class employeeInfo():

    def __init__(self):
        self.L = []

    def __repr__(self):
        s = ""
        for x in self.L:
            s = s+str(x[0])+" "+str(x[1])+" "+str(x[2])+" "+str(x[3])+"\n"

        return s.strip()
```

Your task for this problem is to implement the `add` method.
(**Hint:** The implementation of the `__repr__` shown above contains many clues about how the list of employee records is organized.)