

Fall 2022 In-class Midterm Exam Overview

On Tuesday or Wednesday, Nov 8 and 9, 2022, you will take a midterm exam during your CS111 lab. At the end of this document is a list of topics for the exam.

Your exam will be hand-written by you on paper. Each question has boxes where you will be asked to write your answer. Only write inside the boxes. We will scan your exam into Gradescope and will grade what you write within the boxes only (so do not write in the margins or outside the boxes).

In the exam you could be asked to:

1. Read Python programs and explain what they do. What values do they print or return?
2. Modify existing Python programs.
3. Write Python programs that satisfy a specification.

The exam is open notes in the sense that you can bring with you any **printed and handwritten materials**, such as your written notes, printouts of slides and web pages you think are important, and books. We strongly recommend against printing large numbers of pages, since most students don't have time during the exam to consult them. [To prepare for the exam, it's better for you to write a few pages of your own notes of what you think is important and might forget.](#)

You are **not** allowed to use any devices during the exam, including but not limited to computers, calculators or smartphones. You are **not** allowed to browse the web during the exam nor use a Python interpreter during the exam.

Here are some things we encourage you to do to prepare for the exam:

- **Practice solving lots of problems involving Python concepts and coding.**
Where can you find such problems?
 - This document has several problems from past midterms and quizzes
 - Redo problems from quizzes (we have provided empty quiz documents)
 - Problems from the lecture notebooks (these were optional so far)
 - Redo problems from the Exercises notebooks (download fresh copies of them from the schedule and go over the ones that you found challenging the first time around)
- Review the quizzes and corresponding solutions (website)
- Review the posted solutions for all project tasks. Often, the posted solutions may show you how to solve a problem in a better way than you did. You can access the solutions from your Potluck account.
- Review all course lecture slides, notebooks and lab materials. Write down anything you're confused about and ask an instructor/tutor.

Concepts for Midterm Exam

The exam covers all material in the course up to and including Lec 14 (Lists Comprehension & Sorting), Lab 7 (Nested Loops), and Project 7.

- Python syntax: expressions vs. statements vs. declarations.
 - Expressions are program fragments that denote values. They may be arbitrarily complex, and are evaluated left-to-right, from the inside out.
 - Statements are program fragments that perform actions. They are composed in chunks that are executed from top down.
 - Declarations introduce variables and function definitions.
- Variables and assignments.
 - To evaluate the assignment expression `<var> = <exp>`, first evaluate `<exp>` to a value `V`.
 - If `<var>` does not yet exist, create a box labeled `<var>` in the current scope (*local*: inside a function, or *global*: the entire program), and fill it with the value `V`.
 - If `<var>` already exists, change the contents of the box labeled `<var>` to the value `V`.
 - In the context of assignments involving list slots, `<var>` can be replaced by an expression denoting a list slot. For example

```
myList[3] = 17 or  
myList[i+1][j-1] = myList[i][j] + myList[i+1][j]
```
 - To evaluate the variable reference expression `<var>`, return the contents of the variable box labeled `<var>` in the current scope.
- Functions:
 - understanding the difference between function definition and function invocation.
 - function parameters
 - The name of parameters does not matter as long as they are used consistently.
 - In a function invocation frame, each parameter denotes a local variable initialized to the argument value.
 - understanding the difference between **return** and **print**.
- Scope:
 - the locality of parameters and other local variables assigned within a function body.
- Booleans/Predicates/Conditionals:
 - Boolean values are just `True` and `False`.
 - Logical operators that operate with boolean values: **not**, **and**, **or**
 - A predicate is just a function that returns a boolean.

- It is often the case that the bodies of predicate functions can be written without **if** statements by using boolean expressions.
 - Simple **if** statement with optional **else** clause.
 - Chained (multibranch) **if** statement with **elif** and **else** clause.
 - Can have nested **if** statements. How are these similar to/different from chained (multibranch) conditionals?
 - Can have sequences of if statements. How do these differ from chained/multibranch conditionals?
 - Consider drawing a flowchart (diagram with diamonds for conditionals and arrows to indicate control flow)
- Sequences:
 - Strings, lists, and tuples are sequences. Their items can be indexed via indices that start at 0. The slice operator `:` can return a subsequence. The subsequence is copied, not aliased.
 - Function **range** creates lists of integer numbers, useful for indexing sequences.
 - lists are mutable sequences of values. You can both change indexed slots and **append** and **pop** indexed slots from a list.
 - strings are immutable sequences of characters.
 - tuples are immutable sequences of values.
 - Iteration:
 - Iterations are repeated updates to state variables, as expressed in iteration tables via iteration rules
 - Iterations are expressed in Python using loops:
 - **while** loops
 - **for** loops range over sequences and are just **while** loops in disguise
 - **for** loops can be value loops (iteration over the elements of a sequence) or index loops (iteration over the integer indices of the sequence)
 - loop gotchas:
 - premature **return** from sequence
 - updates to state variables in wrong order
 - Sometimes you **want** to return early from a sequence via **return** or **break**
 - It is common to nest one loop within another
 - It is common for one state variable to be an accumulation variable (a “bucket”) that starts containing no information, but is updated to contain more information as the iteration progresses, until it contains all desired information by the end of the loop. Some examples
 - A numeric accumulator variable is initialized to 0, and numbers are added to it during the loop.

- A string accumulator variable is initialized to the empty string, and strings are concatenated with it during the loop.
 - A list accumulator variable is initialized to the empty list, and elements are appended to this list during the loop. In this case the value in the variable is the same list object during the whole loop, but since that list object is mutable, it can change over time.
 - One very common list accumulation pattern is the **mapping pattern**, in which an n-element input list [v_0, v_1, ... v_n-1] is mapped to an n-element output list [f(v_0), f(v_1,) ... f(v_n-1)] for some function f.
 - Another very common list accumulation pattern is the **filtering pattern**, in which an n-element input list [v_0, v_1, ... v_n-1] is mapped to an output list that contains only those elements of the input list that satisfy a predicate p, in the same order that they appear in the input list. So the output list will have between 0 and n elements.
- Understanding how to use functions and objects from their contracts.
 - [turtle](#) has lots of objects with contracts that you've used.
 - you've also seen contracts for operations on sequences (lists and strings) (e.g., how to use sorted or insert, etc.)
 - Problem solving strategies:
 - Divide/conquer/glue
 - Designing iterations (loops) with iteration tables and iteration rules
 - Patterns for iteration and list processing: accumulation, mapping, filtering
 - Incremental programming
 - Memory Diagrams
 - Memory Diagrams show the state of a program involving variables and values, including mutable values like lists and objects.
 - They are especially important for understanding:
 - assignment: changing the value stored in a variable, list slot, or object instance variable.
 - adding or removing slots in a list.
 - aliasing: the same mutable value can be accessed by multiple paths.
 - == vs. is for testing value equality.
 - Be able to translate from a given memory diagram to python code that produces such a diagram and also, given python code, draw the corresponding memory diagram to reflect the state of the variables and values.
 - List comprehension: a compact construct for constructing lists via mapping or filtering
 - Sorting: sorting simple or complex sequences, including the use of **key** functions.

Practice Problems from old midterms and quizzes

All these problems have been part of either midterm exams or quizzes in past semesters. A typical midterm has 6-7 problems (each of them might have several parts). There are more problems here, just for practice purposes. In a midterm exam, we expect all students to do almost all problems.

For this material, if you are spending more than 10-15 minutes on a problem, move on, and return once you have a better understanding of the concepts or the problem solving patterns. Ask questions in Sakai, visit the office hours or your instructors, or stop by the drop-in hours with our tutors.

Problem 1: Mystery while loop

Study the **mystery** function below, which uses the provided **isVowel** function.

```
def isVowel(char):  
    return len(char) == 1 and char.lower() in 'aeiou'  
  
def mystery(word, bound):  
    """Docstring withheld."""  
    result = ''  
    i = 0  
  
    while len(result) < bound and i < len(word):  
  
        if (not isVowel(word[i])) and word[i] not in result:  
            result += word[i]  
            i += 1  
  
    if result == '':  
        return 'No result'  
    return result
```

Predict the outcome of the following invocations of the **mystery** function:

| Function call | Value returned by function call |
|-----------------------|---------------------------------|
| mystery('coconut', 1) | |
| mystery('coconut', 4) | |
| mystery('apple', 2) | |

```
mystery('oooooh', 2)
```

Problem 2: List processing

Below define a function `check` that takes two parameters: 1) a word and 2) a list of words and **returns** the list containing all the words that are alphabetically before the given word.

Here are some example calls of this function and their expected results.

| Function call | Value returned by function call |
|--|---|
| <code>check('candy', ['bear', 'apple', 'donut', 'cave'])</code> | <code>['bear', 'apple']</code> |
| <code>check('cook', ['bear', 'apple', 'donut', 'cave'])</code> | <code>['bear', 'apple', 'cave']</code> |
| <code>check('egg', ['bear', 'apple', 'donut', 'cave'])</code> | <code>['bear', 'apple', 'donut', 'cave']</code> |
| <code>check('ant', ['bear', 'apple', 'donut', 'cave'])</code> | <code>[]</code> |
| <code>check('best', ['baby', 'butter', 'bear', 'beast', 'boo'])</code> | <code>['baby', 'bear', 'beast']</code> |

Type your code inside the box

Problem 3: Loop with conditionals

Below define a function `pigLatin` that accepts a *list* of words and returns a *list* of those same words translated into "Pig Latin." "Pig Latin" is a made-up language that involves shifting letters of a word around and appending the sound "ay."

Here are our rules for this language:

- Words that are shorter than 3 characters are left as is e.g. 'an' => 'an'
- Words that begin with a consonant shift the first letter to the end and append 'ay' e.g. 'hello' => 'ellohay'
- Words that begin with vowels get 'ay' appended e.g. 'apple' => 'appleay'

Here are some example calls of this function and their expected results

| Function call | Result |
|--|--|
| <code>pigLatin(['this','is','a','great','example'])</code> | <code>['histay', 'is', 'a', 'reatgay', 'exampleay']</code> |
| <code>pigLatin(['is'])</code> | <code>['is']</code> |
| <code>pigLatin(['great'])</code> | <code>['reatgay']</code> |
| <code>pigLatin(['example'])</code> | <code>['exampleay']</code> |

Complete the definition of the **pigLatin** function below. Your function must use either a **for** loop or a **while** loop. You may use **isVowel** or other helper functions, though you don't need to.

(Please keep all your code within the box)

Problem 4: Understanding conditionals

In the table below, show what is printed for various calls of this **analyze** function:

```

def analyze(word):
    if len(word) <= 4:
        print('S')
    else:
        print('L')
    if isVowel(word[0]):
        print('V0')
        if not isVowel(word[1]):
            print('C1')
    elif isVowel(word[1]):
        print('V1')
    else:
        print('C01')
    if isVowel(word[-1]): # last letter of word
        print('VU')
        if not isVowel(word[-2]): # next to last letter of word
            print('CP')

def isVowel(char):
    return char.lower() in 'aeiou'

```

| Function call | Printed Output | | Function call | Printed Output |
|-----------------|----------------|--|------------------|----------------|
| analyze('cat') | | | analyze('spree') | |
| analyze('oats') | | | analyze('apple') | |

Problem 5: Printing Time

On the next page, define a function **printTime** that takes three arguments:

1. **day**: a day of the week, which is one of the strings **'Sun'**, **'Mon'**, **'Tue'**, **'Wed'**, **'Thu'**, **'Fri'**, **'Sat'**
2. **hour**: an integer between 1 and 12, inclusive
3. **ampm**: one of the strings **'AM'** or **'PM'**

printTime prints *exactly one word* as specified below. It does not return anything.

- For a weekend day (Sat or Sun), it prints **weekend**.
- For a weekday (Mon through Fri):
 - It prints **evening** from 5PM up to and including 11PM
 - It prints **sleep** from midnight (12AM) up to and including 8AM.
Note that midnight is considered the beginning of a new day, not the end of a previous day.
 - It prints **class** for all other times — i.e., from 9AM up to and including 4PM.
This range includes noon (12PM).

Here are some examples:

| Function call | Printed Output | | Function call | Printed Output |
|---------------------------------------|----------------|--|---------------------------------------|----------------|
| <code>printTime('Sat',12,'AM')</code> | weekend | | <code>printTime('Mon',12,'AM')</code> | sleep |
| <code>printTime('Sat',10,'AM')</code> | weekend | | <code>printTime('Wed',3,'AM')</code> | sleep |
| <code>printTime('Sun',11,'PM')</code> | weekend | | <code>printTime('Fri',8,'AM')</code> | sleep |
| <code>printTime('Mon',5,'PM')</code> | evening | | <code>printTime('Tue',9,'AM')</code> | class |
| <code>printTime('Thu',8,'PM')</code> | evening | | <code>printTime('Wed',12,'PM')</code> | class |
| <code>printTime('Fri',11,'PM')</code> | evening | | <code>printTime('Thu',4,'PM')</code> | class |

In your definition you do **not** need to handle cases where an input is an unexpected value (e.g., an invalid day or ampm string or an hour that is not an integer in the range 1 to 12 inclusive).

(Please keep all your code within the box)

Problem 6: Strings & Loops

Define a function **block**(width, string) that prints a string with width characters per line. Below are some sample invocations. Hint: you might find the function `range()` and slicing helpful.

| | |
|---|--|
| block (4, 'abcdefghijklmnopqrstuvwxyz') | abcd efgh ijkl mnop qrst uvwx yz |
| block (10, 'abcdefghijklmnopqrstuvwxyz') | abcdefghij klmnopqrst uvwxyz |
| block (3, 'THANK YOU') | THA NK YOU |

Write your **block** function here (keep all code within the box below):

Problem 7: Iteration Table (old quiz problem)

For the following function:

```
def divisibleBy(stop, el):  
    divList = []  
    i = 0  
    while i < stop:  
        if i % el == 0:  
            divList.append(i)  
        i += 1  
    return divList
```

In the box at right, write the iteration table that captures how its state variables change for the function call:

```
divisibleBy(9, 3)
```

Problem 8: Selective Summing [Challenging]

Below define a function `sum78` that takes a list of numbers and returns the sum of the numbers in the list, ignoring sections of numbers starting with a 7 and extending to the next 8 (or to the end of the list, if there is no corresponding 8). Return 0 when no numbers are summed.

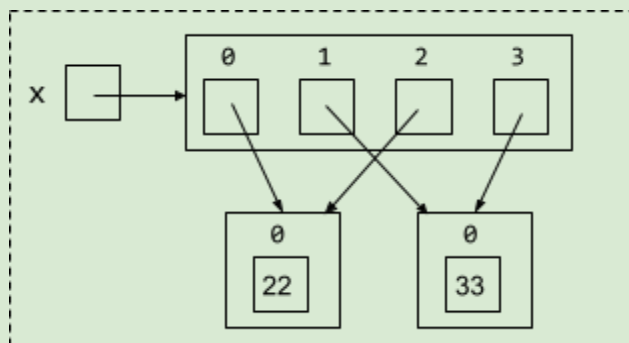
Here are some example calls of this function and their expected results. Numbers with a gray background are ignored.

| Function call | Value returned by function call |
|--|---------------------------------|
| <code>sum78([1, 4, 2])</code> | $7 = 1 + 4 + 2$ |
| <code>sum78([1, 4, 2, 7, 77, 54, 8, 5])</code> | $12 = 1 + 4 + 2 + 5$ |
| <code>sum78([1, 7, 17, 8, 2, 7, 23, 42, 8, 3, 7, 91, 8, 4])</code> | $10 \# 1 + 2 + 3 + 4$ |
| <code>sum78([9, 7, 2, 7, 2, 8, 3, 4])</code> | $16 \# 9 + 3 + 4$ |
| <code>sum78([4, 1, 7, 2, 7, 2, 8, 5, 2, 7, 10, 20, 30])</code> | $12 \# 4 + 1 + 5 + 2$ |
| <code>sum78([7, 6, 1, 6, 8])</code> | 0 |

Problem 9 : Memory Diagrams

Part a: Matching Code Snippets to Memory Diagrams

Below are 20 code snippets that attempt to create the following memory diagram:



Almost all of these snippets were submitted by students as solutions to this problem on a midterm exam during a previous semester. (In some cases, they have been edited in minor ways.)

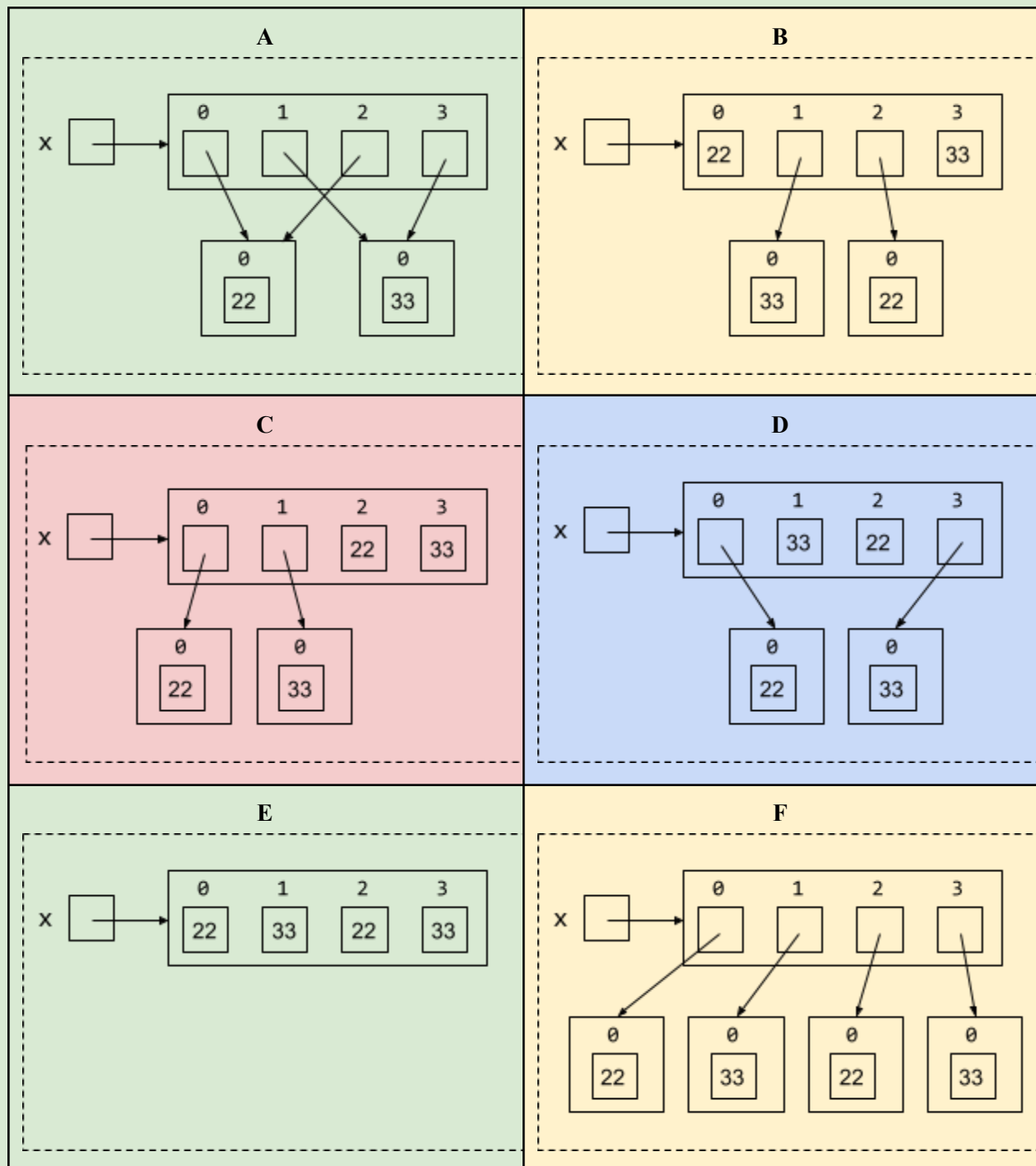
Your goal is to match each of the 20 code snippets with corresponding memory diagrams that are shown below the snippets. (For convenience, the memory diagrams are also available [on this other page](#) so you can view them side-by-side with the snippets.) Each of the code snippets should generate one of the 10 diagrams. Which one? The same diagram may be generated by multiple snippets.

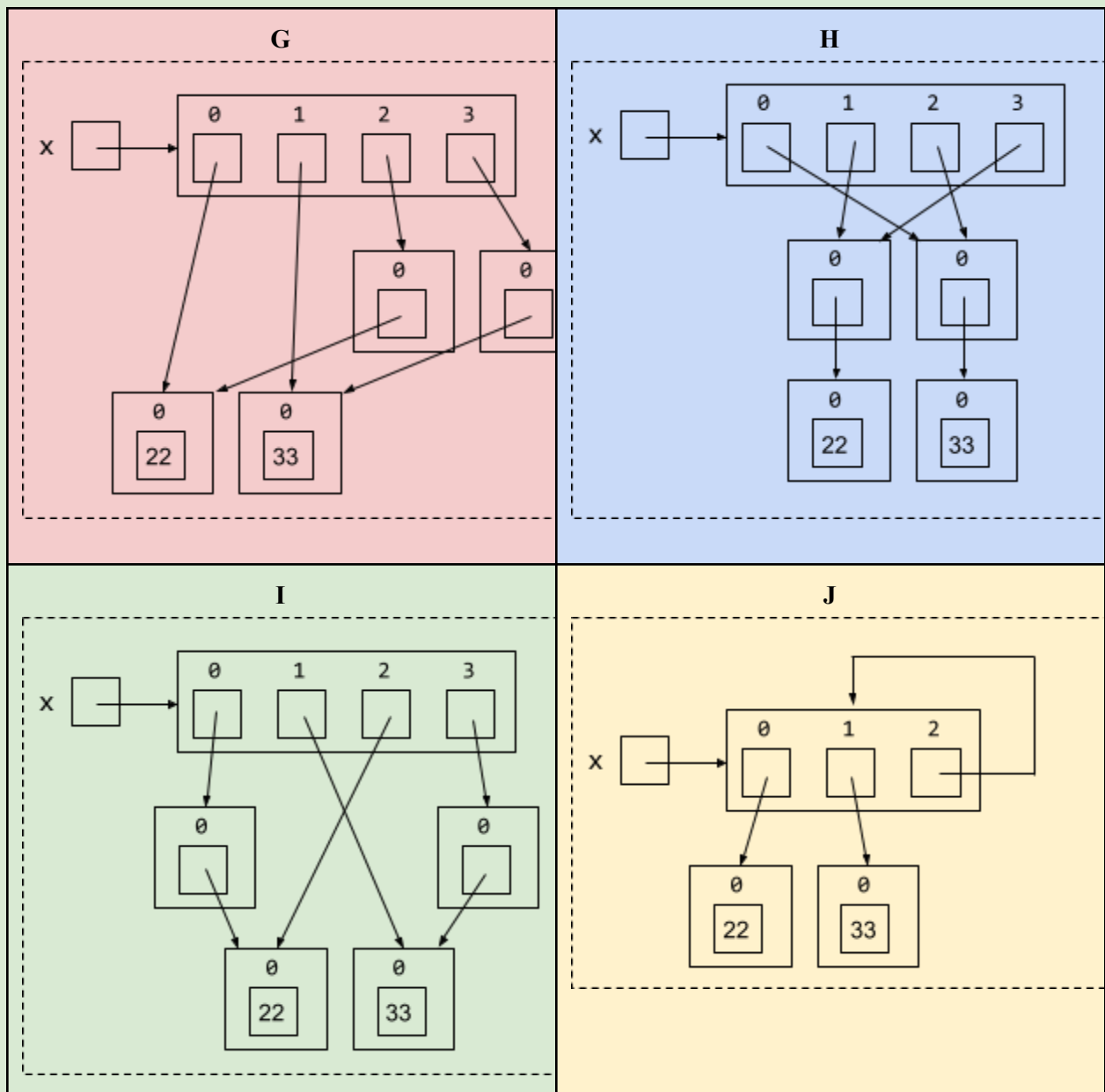
Code Snippets

| | | |
|--|--|--|
| CODE01 <code>a = [22]</code> | CODE02 <code>x = [[22], [33], [22], [33]]</code> | CODE03 <code>x = [[22], [33], [22], [33]]</code> |
|--|--|--|

| | | |
|--|--|--|
| <pre>b = [33] x = [a, b, a, b]</pre> | <pre>x[0][0] = x[2] x[3][0] = x[1]</pre> | |
| CODE04 <pre>x = [[22], 5, 6, [33]] x[2] = x[0] x[1] = x[3]</pre> | CODE05 <pre>x = [[22], [33], [22], [33]] x[2][0] = x[0] x[3][0] = x[1]</pre> | CODE06 <pre>x = [[22], [33], [22], [33]] x[2] = x[0][0] x[3] = x[1][0]</pre> |
| CODE07 <pre>x = [[22], [33], [22], [33]] x[0] = x[2] x[1] = x[3]</pre> | CODE08 <pre>x = [[22], [33], [22], [33]] x[0][0] = x[2][0] x[1][0] = x[3][0]</pre> | CODE09 <pre>x = [[22], [33], 5, 6] x[2] = x[0] x[3] = x[1]</pre> |
| CODE10 <pre>x = [[22], 5, 6, [33]] x[1] = x[3][0] x[2] = x[0][0]</pre> | CODE11 <pre>x = [[22], [33]] x = x*2</pre> | CODE12 <pre>r = [[22]] s = [[33]] x = [s, r, s, r]</pre> |
| CODE13 <pre>x = [22, 33] x.append(x[0]) x.append(x[1])</pre> | CODE14 <pre>x = [[22], [33]] x.append(x[0]) x.append(x[1])</pre> | CODE15 <pre>x = [[22], [33]] x.append(x)</pre> |
| CODE16 <pre>x = [[22], [33]] x.append(0) x.append(1) x[2] = x[0] x[3] = x[1]</pre> | CODE17 <pre>x = [[33], [22]] x.insert(0, x[1][0]) x.append(x[1][0])</pre> | CODE18 <pre>x = [[22], [33]] x.insert(1, x[1][0]) x.insert(2, x[0][0])</pre> |
| CODE19 <pre>x = [[22], [33], [22], [33]] x[2] = x[0] x[3] = x[1]</pre> | CODE20 <pre>x = [[[22]], [[33]], [[22]], [[33]]] x[1] = x[3][0] x[2] = x[0][0]</pre> | |

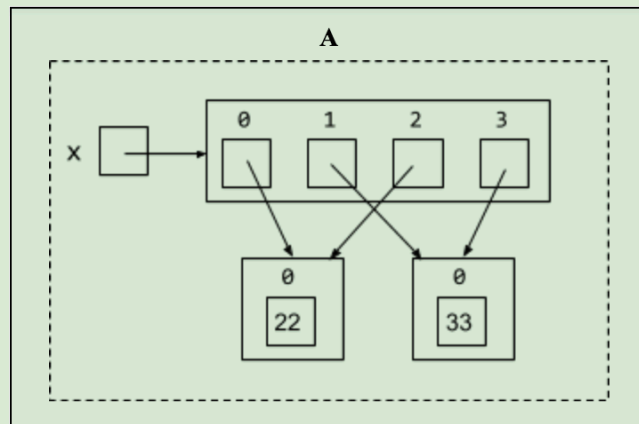
Memory Diagrams



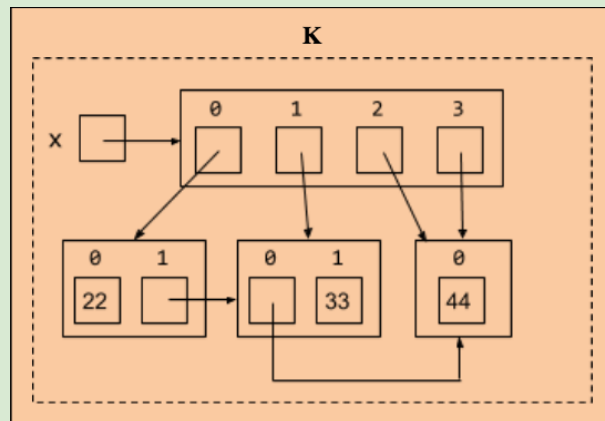


Part b: Modifying Memory Diagram A

Assume that the value of variable x is the list with memory diagram A:



In this part, you will flesh out the body of the function `A_to_K` that takes as its single parameter a list named `L` and modifies it in such a way that after calling `A_to_K(x)`, the structure of `x` has been changed to that of memory diagram **K**

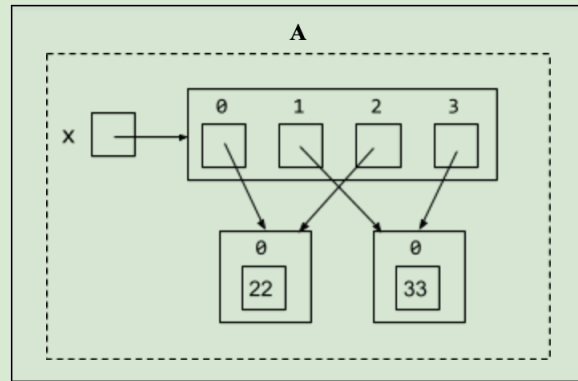


In the body of the `A_to_K` function, your code should satisfy the following restrictions:

- The **only** local variable it is allowed to use is the parameter `L`. It must **not** refer to the variable `x`, nor should it introduce any other local variable names. (But because you may assume that the `A_to_K` function is called on `x`, `x` and `L` will point to the very same list.)
- It must create exactly **one** new list, the singleton list whose only list slot contains 44.
- It must **not** change the contents of any existing list slot containing a number. So the list with a slot containing 22 in diagram **K** must be a modified version of the list with the slot containing 22 in diagram **A**, and similarly for the list containing 33.
- It **must** change the contents of some existing list slots that do not contain numbers.
- Your code may create new list slots in existing lists by calling the `.append` and `.insert` methods.

Part c: Drawing a Modified Memory Diagram

Assume that the value of variable `x` is the list with memory diagram **A**:



In this part, draw the **final** memory diagram that shows the structure of the list `x` after executing the following sequence of statements:

```
x[3] = [x[1], x[0], x[2][0]]
x[1].insert(0, x[2])
x[0].append(56)
x[2][0] = x[2][1] + x[1][1] + x[3][0][0][0]
```

Problem 10 : More Lists Operations

Part A: Given snippets of code that represents a list comprehension, translate the code to normal for loops syntax.

```
highFives = [num*5 for num in range(21) if num % 5 == 0]
```

Part B: Rewrite the solution of Problem 2 (List Processing) on page 6 using list comprehension. That is, rewrite the function `check` to have a single line that is a list comprehension.

