NAME ________________________________ **ICA-12**

NetID______________________ @email.arizona.edu

Work with your neighbor.

Problem 1. Write a Python function `times_i_at_odd(L)` that takes as arguments a list L and returns a list consisting of the elements of L multiplied by the index number of the element at odd positions. (Use list comprehensions)

```
>>> times_i_at_odd([1,2,3,4,5,6,7,8,9,10])
[2, 12, 30, 56, 90]
```

Problem 2. Write a recursive function `sum_cols(grid, n)` that takes a list of lists of integers `grid` and integer `n` and returns the sum of column `n` in `grid`. For example, the call

```
    sum_cols([[1,2,3,4], [10,20,30,40], [100,200,300,400]], 2)
```

should return `333`.

Problem 3. Write a recursive function `replace(s, a, b)` that takes as arguments the strings `s`, `a`, and `b` and replaces all the occurrances of string `a` with string `b` in string `s`.

a) For your solution, assume that string `a` is only one character long.

b) For your solution, assume that string a can be of any size. Use `s.startswith(a)`, which returns `True` if string `s` starts with `a` and `False` otherwise.

Problem 4.

a) For this problem, assume the following implementation of a queue.

```
class Queue:
    def __init__(self):
        self._items = []

    def enqueue(self, item):
        self._items.append(item)

    def dequeue(self):
        return self._items.pop(0)

    def __str__(self):
        return str(self._items)
```

Given the statement

```
q = Queue()
```

write what `print(q)` would output after each of the statements below:

```
q.enqueue(10)          ______________________

q.enqueue(20)          ______________________

q.enqueue(30)          ______________________

q.dequeue()            ______________________
```

b) Modify the implementation of the `Queue` class such that the capacity of the queue is specified when the queue is created. If an enqueue operation would cause the capcity to be exceeded, the enqueue operation has no effect on the queue, i.e., the queue is not modified and no error is produced. Re-write only the methods that are affected by these requirements.

Problem 5.

a) Some calculators use postfix notation for arithmetic expressions. In this notation, operators come after their operands. For example, the expression

   (5 + 2) * 3 – 7

   would be represented in postfix notation as

   5 2 + 3 * 7 –

   Postfix expressions can be evaluated using a stack as described below. This assumes that the expression has been parsed into a sequence of *tokens*, where a **token** is either a number or an operator:

```
S = Stack()
for token in expression:
   If token is a number
      push token on the stack S
   else (the token is an operator)
      Let op be the operator specified by token
      pop two values v1 and v2 from S
      evaluate the expression v1 op v2
      push the result on the stack S
   <show the stack>

#at the end there should be exactly one number on the stack
pop a value v from S and return v
```

   Work through this algorithm and evaluate the expression above. After each iteration, draw the stack on this sheet of paper.

b) We can easily write function to evaluate an arithmetic expression in postfix notation using the algorithm above. There are only two details that need to be addressed.

- Parsing: It will be straightforward to "parse" the expression if the expression is a string separated by spaces. We will use split() to produce a list of string tokens.

- Performing the operation: The other detail is the matter of performing the operation once an operator is encountered. If v1 and v2 are values popped off of the stack, and the token is "+", how do we get Python to perform the addition? One way to do this is to import the operator module and use the following functions from the module. The functions can be put into a dictionary for easy access as shown below:

```
>>> from operator import *
>>> add(3,4)
7
>>> sub(8,2)
6
>>> mul(2,5)
10
>>> truediv(9,3)
3.0
>>> ops = {"+": add, "-": sub, "*": mul, "/": truediv}
>>> ops["+"](3,4)
7
```

Write a function `eval_postfix(expr)` that takes a string `expr` which is an arithmetic expression in postfix notation, evaluates the expression using the algorithm in a), and returns the result.

```
>>> eval_postfix("5  2  + 3  * 7 - ")
14
>>> eval_postfix("5  2  + 3  / 7 - ")
-5
```