- This lab will cover Stacks.
- You may want to refer to the text and your lecture notes during lab as you solve the problems.
- When approaching the problems, think before you code. It is good practice and generally helpful to lay out possible solutions for yourself.
- You should write test code to try out your solutions.
- You must stay for the duration of the lab. If you finish early, you can help other students to complete the lab. If you don't finish by the end of the lab, it is recommended that you complete the lab on your own time.
- Your TAs are available to answer your questions in lab, during office hours, and on Piazza.

Vitamins (maximum 30 minutes)

---

1. What is the output of the following code?

```
import ArrayStack

s = ArrayStack.ArrayStack()
i = 2

s.push(1)
s.push(2)
s.push(4)
s.push(8)

i += s.top()
s.push(i)
s.pop()
s.pop()
print(i)
print(s.top())
```

2. Trace the output of the function with the following call. Draw the state of both stacks as the function executes.

```
calculate("( ( ( 60 + 40 ) / 50 ) * ( 16 - 4 ) )")
```

```
from ArrayStack import *

def calculate(string_input):
    numbers = ArrayStack()
    operators = ArrayStack()
    symbols = {"+", "-", "*", "/"}
    expression_list = string_input.split()

    for item in expression_list:
        if item.isdigit():
            numbers.push(int(item))
        elif item in symbols:
            operators.push(item)
        elif item == ")":
            # evaluate
            rhs = numbers.pop()
            lhs = numbers.pop()
            operation = operators.pop()
            if operation == "+":
                res = lhs + rhs
            elif operation == "-":
                res = lhs - rhs
            elif operation == "*":
                res = lhs * rhs
            elif operation == "/":
                res = lhs / rhs
            numbers.push(res)
    return numbers.top()
```

Coding

In this section, it is strongly recommended that you solve the problem on paper before writing code.

1. In an HTML document, portions of text are delimited by HTML tags. A simple HTML tag has the form <name> and the corresponding closing tag has the form </name>. Ideally, an HTML document should have matching tags. Write a function to check whether html tags are matched/nested properly for the html file provided with the lab.

    a. First, write a generator `get_tag()` that returns the next html tag from some text. You can assume that there are no < or > signs in the text, other than around tags. You many want to refer to the `get_token()` or `tokens()` method implemented during lecture.
    b. Write an `is_matched_html(expr)` method that checks whether the html tags are matched. Hint: use the code below which checks for matched expressions as a guide:

```
def is_matched(expr):
    lefty = '({['
    righty = ')}]'

    S = ArrayStack()

    for token in expr:
        if token in lefty:
            S.push(token)
        elif token in righty:
            if S.is_empty( ):
                return False
            from_S = S.top()
            if (righty.index(token) == lefty.index(from_S)):
                S.pop()
            else:
                return False
        else:
            raise ValueError

    return S.is_empty()
```

2. Implement a function `eval_postfix_boolean_exp(boolean_exp_str)` that takes a string containing a boolean postfix expression and evaluates the boolean result of the expression. The input string will contain a single `'&'` to represent AND and a single `'|'` to represent OR. You can assume that all operands and operators are separated by spaces.

   For example, the expression `'5 2 <'` would return False while the expression `'2 5 <'` would return `True`. Similarly, the expression `'1 2 < 6 3 < &'` would return False while `'1 2 < 6 3 < |'` would return `True`.

3. Implement a function `convert_infix_exp_to_postfix(infix_exp_str):` which should take a fully parenthesized infix expression as input and return the postfix version of the expression.

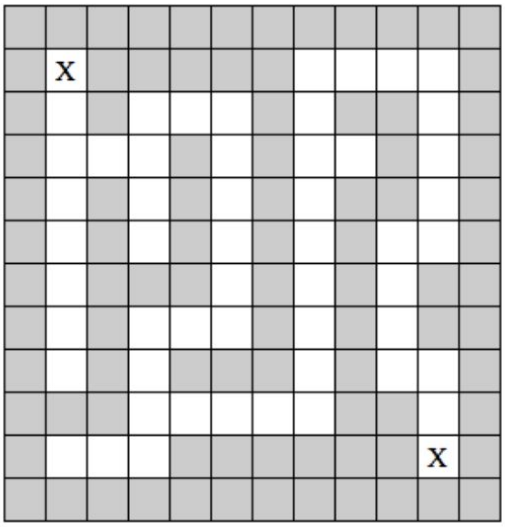
   For example, a call with the expression `'(2 + ((3 * 4)-5)'` should return the postfix expression `'2 3 4 * 5 - +'`.

Extra:

In this question we will write a program that finds a path through a maze, by enumerating possible paths and backtracking when a currently searched path reached a dead end. To implement this kind of a search we will be using a stack.

The maze will be represented as a matrix where a `"1"` indicates a wall that cannot be moved through, and a `"0"` indicates an open path. The graphic on the left shows how the maze can be visualized and the one on the right shows how the maze will be represented by a 0/1 matrix. The start position of the maze it at position [1][1] where the coordinates correspond to the row number and column number. The end position is at [10][10].

Note: In our implementation we have walls in all the outer frame of the maze. That will make your implementation a bit more simple.




```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 0 0 0 0 1
1 0 1 0 0 0 1 0 1 1 0 1
1 0 0 0 1 0 1 0 0 1 0 1
1 0 1 0 1 0 1 0 1 1 0 1
1 0 1 0 1 0 1 0 1 0 0 1
1 0 1 1 1 0 1 0 1 0 1 1
1 0 1 0 0 0 1 0 1 0 1 1
1 0 1 0 1 1 1 0 1 0 0 1
1 1 1 0 0 0 0 0 1 1 0 1
1 0 0 0 1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

First, you will need to read a maze from a file into a two-dimensional array. We have supplied a '.txt' file on NYU Classes. A two-dimensional array is essentially a list of lists. We will visualize the list as a matrix for this problem, but printed in list form it looks like this:

```
[['1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1'], ['1',
'0', '1', '1', '1', '1', '1', '0', '0', '0', '0', '1']....]
```

You can index positions in this matrix with list indexing notation. For example, `maze[5][4]` will return the character at the position row 5 and column 4.

You can use the following code to read in the file and create a matrix:

```
temp_file = open("maze.txt")
lines = temp_file.readlines()
temp_file.close()

maze = []
for item in lines:
   maze.append(item.split())
```

You can use the following code if you would like to print the matrix: (although this is not necessary to solve the problem)

```
for row in maze:
   for item in row:
       print(item, " ", end="")
   print()
```

In order to move through the maze, you will check neighboring positions (up, down, right and left) from your current position to see if they are a valid position to move to (“0”) or a wall which you cannot move through (“1”).

You should use the following class to keep track of your position in the maze. A `MatrixPosition` object will store its row number, column number and the direction that was traveled upon leaving that position.

```
class MatrixPosition():
      def __init__(self, row = 1, col = 1, direction = ""):
          self.row = row
          self.col = col
          self.direction = direction

      def __eq__(self, other):
         return (self.row == other.row and
                 self.col == other.col and
                 self.direction == other.direction)

      def __repr__(self):
          return "(" + str(self.row) + "," + str(self.col) + ")"
```

Before you try to move to a new position, you should check to see if you have reached the end of the maze. Each time you move from a position, you should push the `MatrixPosition` object onto the stack containing information about the position you moved from, and the direction you moved to. Additionally, If you get stuck at a dead end (you are unable to move in any direction except the direction you came from, or directions already searched), you will need to pop that position off the stack, trying to search for another path.

When you reach the end of the maze, output the path from the end back to the beginning and the number of steps that it took to complete the maze. You can use the following function to output your path:

```
def print_path(maze_path):
   counter = 0
   print("Maze Path:")

   while not maze_path.is_empty():
       print(maze_path.top())
       maze_path.pop()
       counter += 1
   print("It took", counter, "steps to complete the maze.")
```