1 Nonlocal

1.1 (a) Draw the environment diagram that results from running the code.

```python
def what(a, b):
    x = a
    def ha(ha):
        nonlocal x
        x = ha * 2
        return x
    return b(ha(x), x)
```

what(4, lambda x, y : x)

[https://goo.gl/aGQF7c](https://goo.gl/aGQF7c)

(b) Write the simplest possible function that does the same thing as `what` for any input `a`, `b`.

```python
def foo(a, b):
    a *= 2
    return b(a, a)
```

1.2 Draw the environment diagram that results from running the code.

```python
def campa(nile):
    def ding(ding):
        nonlocal nile
        def nile(ring):
            return ding
        return nile(ding(1914)) + nile(1917)

ring = campa(lambda nile: 103)
```

[https://goo.gl/G1Kmbw](https://goo.gl/G1Kmbw)
The ping-pong sequence counts up starting from 1 and is always either counting up or counting down.

At element $k$, the direction switches if $k$ is a multiple of 7 or contains the digit 7.

The first 30 elements of the ping-pong sequence are listed below, with direction swaps marked using brackets at the 7th, 14th, 17th, 21st, 27th, and 28th elements.


Implement `make_pingpong_tracker` which returns the next value in the pingpong sequence each time it is called.

```python
def has_seven(n):
    """Returns whether a number, n, contains the digit, 7."""
    if n % 10 == 7:
        return True
    elif n < 10:
        return False
    else:
        return has_seven(n // 10)

def make_pingpong_tracker():
    """
    >>> output = []
    >>> x = make_pingpong_tracker()
    >>> for _ in range(9):
    ...     output += [x()]
    >>> output
    [1, 2, 3, 4, 5, 6, 7, 6, 5]
    """

    index, current, add = 1, 0, True
    def pingpong_tracker():
        nonlocal index, current, add
        if add:
            current = current + 1
        else:
            current = current - 1
        if has_seven(index) or index % 7 == 0:
            add = not add
        index += 1
        return current
    return pingpong_tracker
```
2 Linked Lists & Trees, Yet Again?

2.1 Consider the following linked list function.

```python
def prepend(s, item):
    return link(item, s)
```

(a) What does this function do?

It takes in an existing list and returns a new list with the item at the front.

(b) Assume s is initially length n, how long does it take to prepend once? Prepend twice? Prepend n times?

All inserts will take constant time. No matter how long the list is, it doesn’t take any longer to add to the front. One insert will take one unit of time, and two will take roughly twice that. Therefore, the amount of time to do n inserts will be in $\Theta(n)$.

2.2 What does this function do?

```python
def append(s, item):
    if s is empty:
        return link(item)
    else:
        return link(first(s), append(rest(s), item))
```

append inserted the item at the end of the list in $\Theta(n)$ time where n is the length of the sequence, s.

2.3 Say we want to repeatedly insert some numbers to the end of a linked list.

```python
def append_many(s, items):
    for item in items:
        append(s, item)
```

(a) Assuming s is initially length 1. How long will it take to complete the first insertion? The second? The nth?

Notice that the list gets longer with each insertion, so each operation will make it harder to do the next. Therefore, the first insertion will take about 1 unit of time. The second will take about two units of time. The nth insertion will take n units of time.

(b) Give the total runtime in $\Theta(\cdot)$ notation if s is initially empty and items contains n items.

The total runtime will be the sum of all the inserts: $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} \in \Theta(n^2)$.
2.4 Consider the word_finder function below.

```python
def word_finder(t, n, word):
    if root(t) == word:
        n -= 1
        if n == 0:
            return True
    for branch in branches(t):
        if word_finder(branch, n, word):
            return True
    return False
```

(a) What does this function do?

This function takes a tree \( t \), an integer \( n \), and a string \( \text{word} \) in as input.

\( \text{wordfinder} \) returns true if any paths from the root towards the leaves have at least \( n \) occurrences of the word and false otherwise.

(b) If a tree has \( n \) total nodes, what is the total runtime for all searches in \( \Theta(\cdot) \) notation?

\( \Theta(n) \). At worst, we must visit every node of the tree.