Name: _____

1. Looping

This problem was adapted from UC Berkeley's CS 61B course.

One way to measure the efficiency of a program is by analyzing its ***time complexity***, which describes how the runtime grows as the input size increases. Let `N` be some arbitrary large integer. Suppose that `do_work()` is a function that runs in $O(1)$ time, also known as constant time.

Consider the `for` loop below:

```
for i in range(N):
    do_work()
```

a. What is the time complexity in terms of `N`?

Runtime: $O($_____$)$

b. If we were to change `range(N)` to `range(5 * N)`, would the time complexity change? If so, what would be the new $O(\cdot)$ bound? If not, why not?

c. If the runtime of `do_work()` changed from $O(1)$ to $O(N^2)$, would the overall time complexity change? If so, what would be the new $O(\cdot)$ bound?

Now, consider the nested `for` loops below. Suppose that `do_work()` runs in constant time.

```
for i in range(N):
    for j in range(N):
        do_work()
```

d. What is the time complexity in terms of `N`?

Runtime: $O($_____$)$

Suppose we replace the inner loop with the following loop:

```
j = 1
while j < N:
    do_work()
    j *= 2
```

e. Would the overall time complexity change? If so, what would be the new $O(\cdot)$ bound? If not, why not?

2. Code Fragments

This problem was adapted from UC Berkeley's CS 61B course.

a. Analyze the following code fragments and determine the time complexity of each using O notation with respect to `N`. Assume that `print()` runs in constant time.

```
x = 7
while x < N + 14:
    print("tidal wave")
    x += 1
```

Runtime: $O($_____$)$

```
for x in range(1, N):
    for y in range(1, N // 2):
        print("amethyst")
```

Runtime: $O($_____$)$

```
for i in range(1, N):
    for y in range(1, 99999999):
        print("anathema")
```

Runtime: $O($_____$)$

```
# Assume that numbers is a list of integers.
old_size = len(numbers)

for i in range(old_size):
    x = numbers[0]
    numbers = numbers[1:]
    print(x)
```

Runtime: $O($_____$)$

```
# Assume that numbers is a sorted list of integers.
N = len(numbers)

for i in range(N):
    x = binary_search(numbers, i)
    print(x)
```

Runtime: $O($_____$)$

3. Best and Worst Case

When analyzing algorithms, we often examine how they perform under different conditions. The best case describes the fastest possible completion time, while the worst case describes the slowest possible completion time, often reflecting the most demanding input.

a. Give the best-case and worst-case runtimes in terms of `N`, the length of the sorted list.

```
def mystery(numbers):

    N = len(numbers)

    for i in range(N):

        index = binary_search(numbers, i)

        if index != -1:
            return True

    return False
```

Best Case Runtime: $O($_____$)$

Worst Case Runtime: $O($_____$)$

b. Give the best-case and worst-case runtimes of `until_duplicate(N)`. Assume that `random.randint(a, b)` runs in constant time.

```
import random

def until_duplicate(N):

    seen = []

    while True:

        num = random.randint(0, N - 1)

        if num in seen:
            return num

        seen.append(num)
```

Best Case Runtime: $O($_____$)$

Worst Case Runtime: $O($_____$)$

c. Give the best-case and worst-case runtimes of `find_power(target)` in terms of `N`.

```
def find_power(target):

    x = 1

    while x <= N:

        if x == target:
            return True

        x *= 2

    return False
```

Best Case Runtime: $O($_____$)$

Worst Case Runtime: $O($_____$)$

d. Give the best-case and worst-case runtimes of `remove_until(numbers, target)` in terms of `N`, the length of the list.

```
def remove_until(numbers, target):

    while len(numbers) > 0:

        if numbers[0] == target:
            return True

        numbers = numbers[1:]

    return False
```

Best Case Runtime: $O($_____$)$

Worst Case Runtime: $O($_____$)$

4. Binary Search

Suppose we had the following list: `[1, 4, 6, 8, 10, 12, 13, 16, 20, 25, 30, 36]`.

a. Write the indices that are visited when performing `binary_search(numbers, 25)`.

b. Write the indices that are visited when performing `binary_search(numbers, 7)`.

c. Now, let's try implementing `binary_search` in code.

```
def binary_search(numbers, target):
    left = ____________________
    right = ____________________
    while _______________________________:
        mid = ____________________________
        if ___________________ == ___________________:
            return ___________________
        elif ___________________ < ___________________:
            ___________________ = ___________________
        else:
            ___________________ = ___________________
    return ___________________
```