The goal of this worksheet is to reinforce some common principles of algorithm running time analysis that we have been using all term. Each code snippet/function/method below has some input variables, and performs some operations. For each one, write down beside it its Big-Oh worst-case running time, as well as a brief explanation.

Remember that a **constant time** operation is one whose running time does not depend on the size of the input. All Python arithmetic and comparison operations, variable/attribute access and assignments, and `print` and `return` all take constant time.

---

### # n and m are numbers
```python
print(n)
print(m)
print(n + m)
```

$O(1)$. Each individual line takes *constant time*, and there are a constant number of them.

### # A is a list of numbers of length r
```python
s = 0
y = 10
for x in A:
    s += x
    print(x)
print(s * y)
```

$O(r)$. The loop occurs for $r$ iterations, with each iteration taking constant time.

### # A is a list of numbers of length r
```python
i = 0
s = 0
while i < len(A):
    s += A[i]
    print(A[i])
    i += 10
print(s)
```

$O(r)$. Note that the number of iterations is now $\frac{r}{10}$, but that’s still just a constant factor that is hidden by the Big-Oh.

### # A is a list of length n, B is a list of length m
```python
for x in A:
    print(x)
for y in B:
    print(10)
```

$O(n + m)$ or $O(\max(n, m))$. 

---

CSC148 - Big-Oh Practice
# A is a list of length n, B is a list of length m
for x in A:
    print(x)
    for y in B:
        print(10)

$O(nm)$. The number of iterations of the outer loop is $n$, and each of its iterations takes $O(m)$ time (this is the cost of the inner loop).

# A is a list of length n
for x in A:
    if x == 1:
        print(x)
    else:
        print(x * 2)
        print(x * 3)

$O(n)$. Even though the loop body has an if, the two branches both take constant time, meaning the whole thing will also take constant time.

# a and b are positive numbers
if a > b:
    for i in range(b):
        print(a + i)
else:
    for i in range(a):
        print(b + i)

$O(\min a, b)$. This one is tricky: the two branches take $O(b)$ and $O(a)$ time, so you might think that the running time is $O(\max a, b)$. But the condition tells us that when the first loop runs, $b < a$, and vice versa for the second loop.

# A is a list of length p
for x in A:
    for i in range(10):
        print(x * i)

$O(p)$. The key thing to note is that the inner loop takes a constant amount of time for each iteration of the outer loop.

# A is a list of length n, k is a positive number
for x in A:
    for _ in range(k * k):
        print(x)
# x is a positive number
i = 1
while i < x:
    print(i)
    i *= 2

O(log x). i takes on the values 1, 2, 4, 8, ..., and in general is $2^k$, where $k$ is the current iteration number of the loop. The loop stops when $2^k \geq x$, which is when $k = \log x$.

# A is a list of length n
def find_duplicates(A):
    i = 0
    while i < len(A):
        j = i + 1
        while j < len(A):
            if A[i] == A[j]:
                print('Duplicate found')
                return True
            j += 1
        i += 1
    return False

O($n^2$). This is harder because the number of iterations of the inner loop depends on which iteration of the outer loop occurs. For $i = 0$, the inner loop runs $n-1$ times; for $i = 1$, the inner loop runs $n-2$ times, and so on, until $i = n-1$ and the inner loop runs no times. So the total running time of the inner loop is $(n-1) + (n-2) + \cdots + 2 + 1 + 0 = \frac{n(n-1)}{2}$, which is $O(n^2)$.

# A is a list of length n, k is a positive number
for i in range(k):
    A.append(i)

O(k). There are $k$ loop iterations, and each one takes constant time.

# A is a list of length n, k is a positive number
for i in range(k):
    A.insert(0, i)

O(nk + $k^2$). Tricky! Use the same idea as find_duplicates, except note that the first operation takes $n$ time, and the last operation takes $n + k - 1$ time.