Big O notation describes how algorithmic run time increases with input size. Determine the algorithmic time complexity in Big O notation of the following functions where `values` is **a sorted list** of length `n`

```
def func0(values):
    for i in range(10):
        print(values[-1])
```

O(1)/Constant: We've got a loop, but it runs a fixed number of times, and there's nothing here that depends upon the size of the input.

```
def func1(values):
    n = len(values)
    if n % 2 == 0:
        return (values[n//2-1] + values[n//2])/2
    else:
        return(values[n//2])
```

O(1)/Constant: The stuff in that first return block is pretty messy – you have to count each of the steps separately n//2, (that -1), etc if you want to figure out the precise number of steps, but for asymptotic analysis, we don't care. It will always take the same number of steps.

```
def func2(values, myValue):
    total = 0
    for item in values:
        if item == myValue:
            total += 1
    return total
```

O(n)/Linear: We've got a loop through all of the values in the list, with a constant number of operations for each element, and a constant number of additional operations.

```
def func3(values):
    for i in range(10):
        for j in range(100):
            print(values[-1])
```

O(1)/Constant: We've got two loops, and the inner one runs for quite a few iterations, but it's always the same number of iterations. The input doesn't impact this.

```
def func4(values):
    for i in range(len(values)):
        for j in range(0, len(values), 2):
            print(values[i], values[j])
```

O(n^2)/Quadratic: The outer loop is easy – it clearly runs for N iterations, which contributes a factor of O(n) to the overall cost. The inner loop only runs for half as many iterations, but we ignore constant factors when doing big-O analysis. So, this is the same (asymptotically) as if the inner loop ran for a full N iterations.

These next ones are a bit different – we don't have a list of elements, but a similar idea

```
def func5(n, m):
    for i in range(n):
        for j in range(m):
            print("Hello!")
```

O(n*m)/Linear in N and M: If both loops ran from 0 to n, this would be easy: O(n^2). But, since the second one runs up to n, when we multiply it out, we get O(n*m) instead. We can't simplify this to O(n^2) or O(m^2) because we don't know the values for N or M. N and M could be approximately the same (say, 100, and 117), or they could be much bigger (10 and 10 billion) and we just don't know. This is reasonably common in more advanced algorithms; see f.ex Dijkstra's algorithm.

```
def func6(n, m):
    for i in range(n):
        for j in range(m//2):
            for k in range(10):
                print("Hello!")
```

O(n*m)/Linear in N and M: The inner loop is basically a distraction – we know already that doing 10 times more work doesn't change the picture when doing asymptotic analysis. Otherwise, the explanation is the same as the previous question.