Cookies and Cram: ITI 1120

OOP and Big O review

Code displayed during this session can be found at

github.com/steftodor/ieee-iti1120

Object Oriented Programming

OOP: Point Class

Usage	Explanation
`point.setx(xcoord)`	Sets the x coordinate of `point` to `xcoord`
`point.sety(ycoord)`	Sets the y coordinate of `point` to `ycoord`
`point.get()`	Returns the x and y coordinates of `point` as a tuple `(x, y)`
`point.move(dx, dy)`	Changes the coordinates of `point` from the current `(x, y)` to `(x+dx, y+dy)`

OOP: Point Class – Base Definition

Point.py > ...

```python
class Point:
    'Represents a point in 2-D space'
    def setx(self, x):
        'Set the x coordinate of the point'
        self.x = x
    def sety(self, y):
        'Set the y coordinate of the point'
        self.y = y
    def get(self):
        'Return a tuple representing the point'
        return (self.x, self.y)
    def move(self, dx, dy):
        'Move the point by dx and dy'
        self.x += dx
        self.y += dy
```

All functions in a Class must have a reference to self passed in

OOP: Point Class – Usage

```
>>> from Point import Point # import the class
>>> p = Point()             # create an instance of the class
>>> p.setx(12)              # call the setx method to set the x coordinate
>>> p.sety(1)               # call the sety method to set the y coordinate
>>> p.get()                 # call the get method to get the point
(12, 1)                     # returns a tuple
>>> p.move(2,2)             # call the move method to move the point
>>> p.get()                 # call the get method to get the point
(14, 3)                     # returns a tuple
```

OOP: Point Class – Initial values

```
>>> p = Point(5,5)
```

How?
By overloading the __init__ function

OOP: Point Class – Overloading __init__

Point.py > Point

```python
class Point:
    'Represents a point in 2-D space'

    def __init__(self, xcord, ycord) -> None:
        'Initialize the position of a new point'
        self.x = xcord
        self.y = ycord
    # setx, sety, get, move remain unchanged
```

__init__ is called everytime we create a class

We’re overloading the default __init__ function with one that supports 2 variable inputs

OOP: Point Class – Overloading __init__

```python
def __init__(self, xcord=0, ycord=0) -> None:
    'Initialize the position of a new point'
    self.x = xcord
    self.y = ycord
```

You can set default values if the user doesn't pass in any
If nothing is passed in x and y will both be 0

What other functions can we overload?

OOP: Overloadable Functions

Operator	Method	Number	List and String
x + y	x.__add__(y)	Addition	Concatenation
x - y	x.__sub__(y)	Subtraction	—
x * y	x.__mul__(y)	Multiplication	Self-concatenation
x / y	x.__truediv__(y)	Division	—
x // y	x.__floordiv__(y)	Integer division	—
x % y	x.__mod__(y)	Modulus	—
x == y	x.__eq__(y)	Equal to	
x != y	x.__ne__(y)	Unequal to	
x > y	x.__gt__(y)	Greater than	
x >= y	x.__ge__(y)	Greater than or equal to	
x < y	x.__lt__(y)	Less than	
x <= y	x.__le__(y)	Less than or equal to	
repr(x)	x.__repr__()	Canonical string representation	
str(x)	x.__str__()	Informal string representation	
len(x)	x.__len__()	—	Collection size
<type>(x)	<type>.__init__(x)	Constructor	

OOP: Overloadable Functions

```
>>> p = Point(5,5)
>>> print(p)
```

What happens if we try to print Point p?

OOP: Overloadable Functions

```
>>> p = Point(5,5)
>>> print(p)
<Point.Point object at 0x10308fca0>
```

We have to overload the __str__ function

OOP: Overloadable Functions

Point.py > ...

```python
class Point:
    'Represents a point in 2-D space'

    def __str__(self) -> str:
        'Return a string representation of the point'
        return "Point: ( " + str(self.x) + ", " + str(self.y) + " )"
```

We have to overload the __str__ function

OOP: Overloadable Functions

```
>>> p = Point(5,5)
>>> print(p)
Point: ( 5, 5 )
```

OOP: Inheritance + Overloading

OOP: Inheritance + Overloading

Animals > Animal.py > ...

```python
class Animal:
    'represents a generic animal'

    def setSpecies(self, specs):
        'set the species of the animal'
        self.species = specs
    def setLanguage(self, lang):
        'set the language of the animal'
        self.language = lang
    def speak(self):
        'prints a sentence by the animal'
        print('I am a ' + self.species + ' and I speak ' + self.language)
    def getSpecies(self):
        'returns the species of the animal'
        return self.species
```

OOP: Inheritance + Overloading

Animals > Bird.py > ...

```python
from Animal import Animal
class Bird(Animal):

    def speak(self):
        print('{}!'.format(self.language)*3)
```

Animals > Dog.py > ...

```python
from Animal import Animal
class Dog(Animal):

    def speak(self):
        print(self.language)
```

OOP: Inheritance + Overloading

Animals > Main.py > ...

```python
from Animal import Animal
from Bird import Bird
from Dog import Dog

squirrel = Animal()
squirrel.setSpecies('Flying squirrel')
squirrel.setLanguage('Squeak')
print(squirrel.getSpecies())
squirrel.speak()

parrot = Bird()
parrot.setSpecies('Parrot')
parrot.setLanguage('Squawk')
print(parrot.getSpecies())
parrot.speak()

dog = Dog()
dog.setSpecies('Dog')
dog.setLanguage('Woof')
print(dog.getSpecies())
dog.speak()
```

What is the result?

OOP: Inheritance + Overloading

```
stefan🧸❤️ code/Animals  python3 Main.py
Flying squirrel
I am a Flying squirrel and I speak Squeak
Parrot
Squawk!Squawk!Squawk!
Dog
Woof
```

OOP: Custom Errors

OOP: Custom Errors

Queue > Queue.py > Queue > __str__

```python
from EmptyQueueError import EmptyQueueError
class Queue:
    'a classic queue class'
```

```python
    def dequeue(self):
        'remove and return item at front of queue'
        if self.isEmpty():
            raise EmptyQueueError("dequeue from empty queue")
        return self.q.pop(0)
```

Queue > EmptyQueueError.py > EmptyQueueError

```python
class EmptyQueueError(Exception):
    pass
```

OOP: Custom Errors

Queue > main.py > ...

```python
from Queue import Queue
# initialize a queue
q = Queue()
# enqueue some items
q.enqueue("apple")
q.enqueue("banana")
q.enqueue("cherry")
# print the queue
print(q)
# dequeue some items
print(q.dequeue())
print(q.dequeue())
print(q.dequeue())
# print the queue (should be empty)
print(q)
# try to dequeue from an empty queue (should raise an exception)
print (q.dequeue())

```

```
stefan🧸❤️  code/Queue  python3 main.py
['apple', 'banana', 'cherry']
apple
banana
cherry
[]
Traceback (most recent call last):
  File "/Users/stodorovic/Desktop/iti1120/code/Queue/main.py", line 17, in <module>
    print (q.dequeue())
  File "/Users/stodorovic/Desktop/iti1120/code/Queue/Queue.py", line 16, in dequeue
    raise EmptyQueueError("dequeue from empty queue")
EmptyQueueError.EmptyQueueError: dequeue from empty queue
```

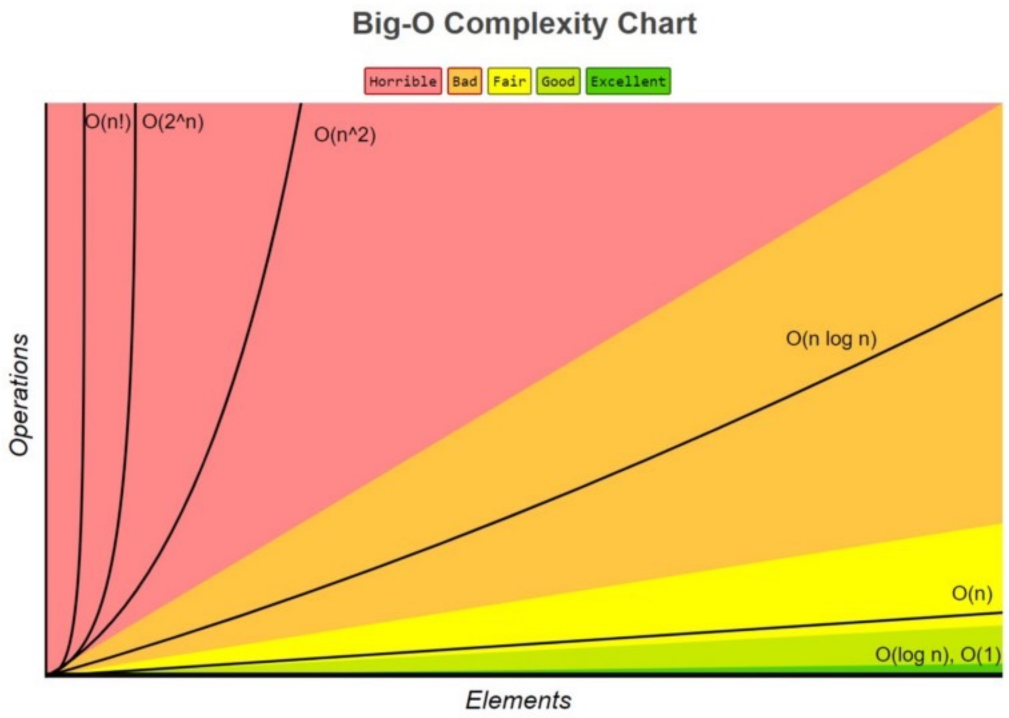
Big O Notation

Big O: What is it?

- The relationship between the number of values entered into an algorithm and the number of steps (or operations) required to complete the algorithm is represented by this variable.
- Is denoted as O(n) where n represents the number of operations

Big O: What is it?

Function Type	BigO
Constant	O(c)
Linear	O(n)
Quadratic	O(n^2)
Cubic	O(n^3)
Exponential	O(2^n)
Logarithmic	O(log(n))
Log Linear	O(n*log*n))



Graph: [freecodecamp]

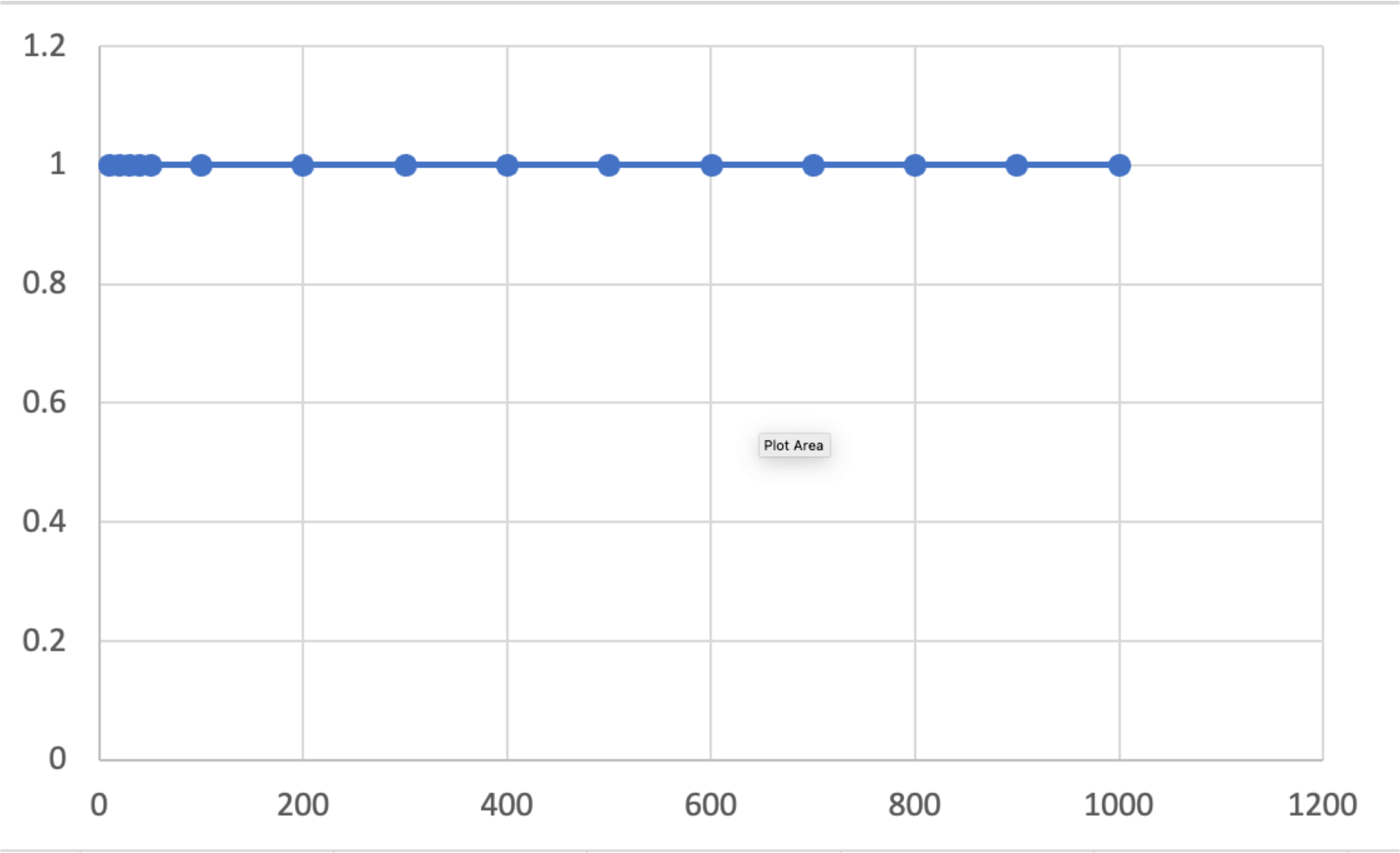
Big O: How do we determine the degree ?

- Refers to the worst case scenario
- Constants are considered negligible
    - Ex. O(2n) -> O(n)

Big O: Constant complexity -> O(c)

```python
def func_constant(values):
    # example of a constant complexity function
    # the number of operations is always the same
    # regardless of the number of items in the list
    temp = values[0]
    var = temp * 20
    ret = var / 3
    return ret
```

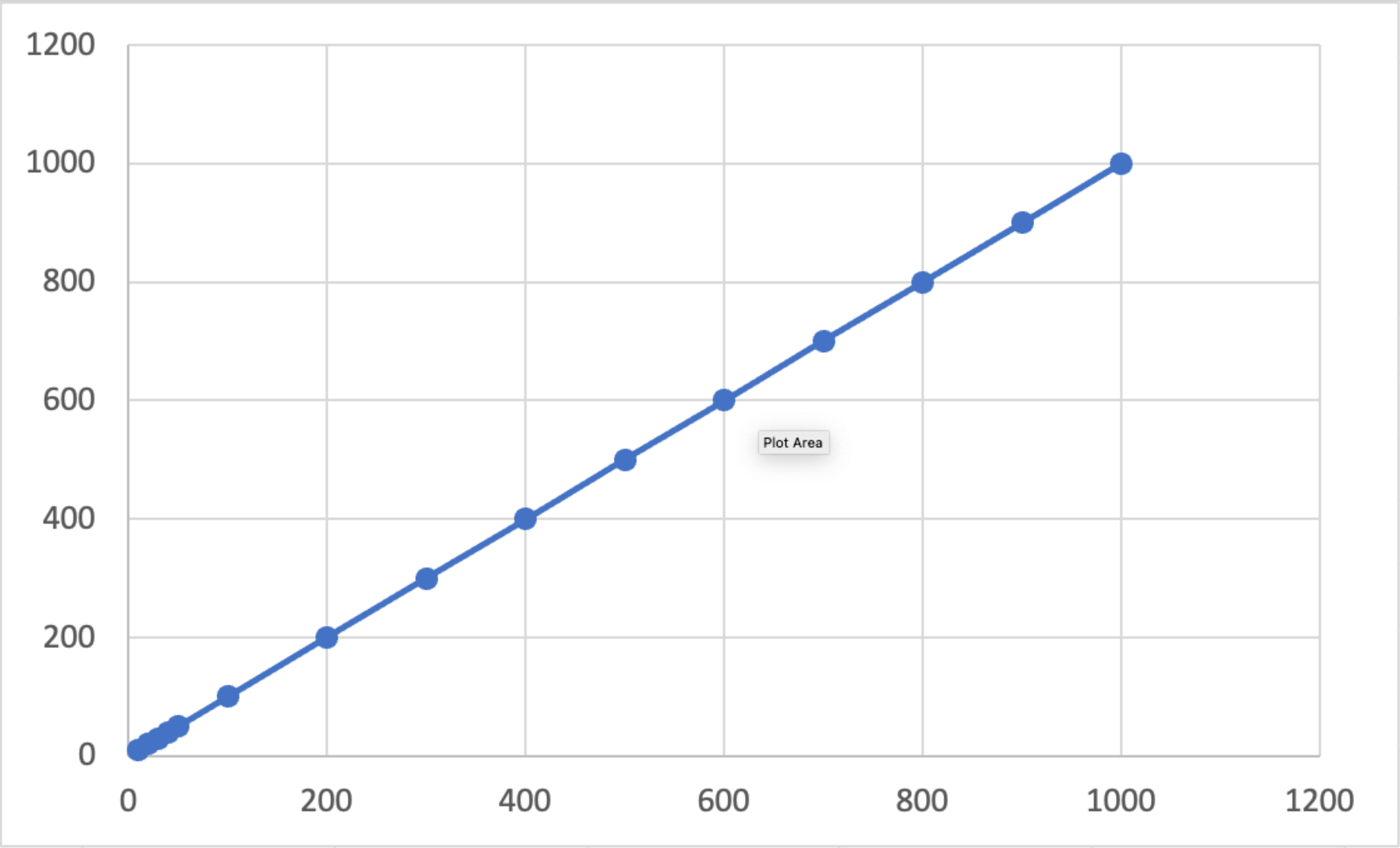
Big O: Constant complexity -> O(c)



Big O: Linear complexity -> O(n)

```python
def func_linear(values):
    # example of a linear complexity function
    # the number of operations is directly proportional
    # to the number of items in the list
    sum = 0
    for val in values:
        sum = sum + val
    return sum
```

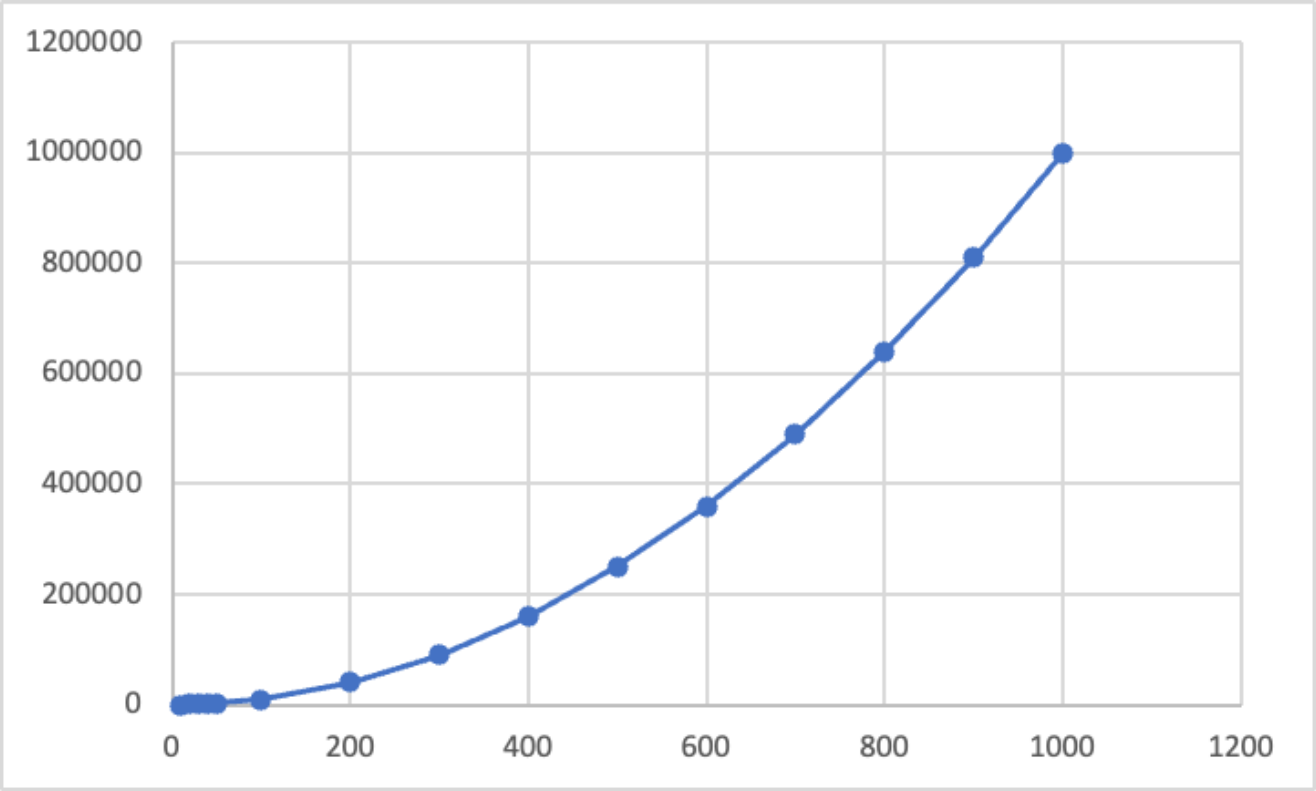
Big O: Linear complexity -> O(n)



Big O: Quadratic complexity -> O(n^2)

```python
def func_quadratic(values):
    # example of a quadratic complexity function
    # the number of operations is proportional to the square of
    # the number of items in the list
    sum = 0
    for val in values:
        for val2 in values:
            sum = sum + val + val2
    return sum
```

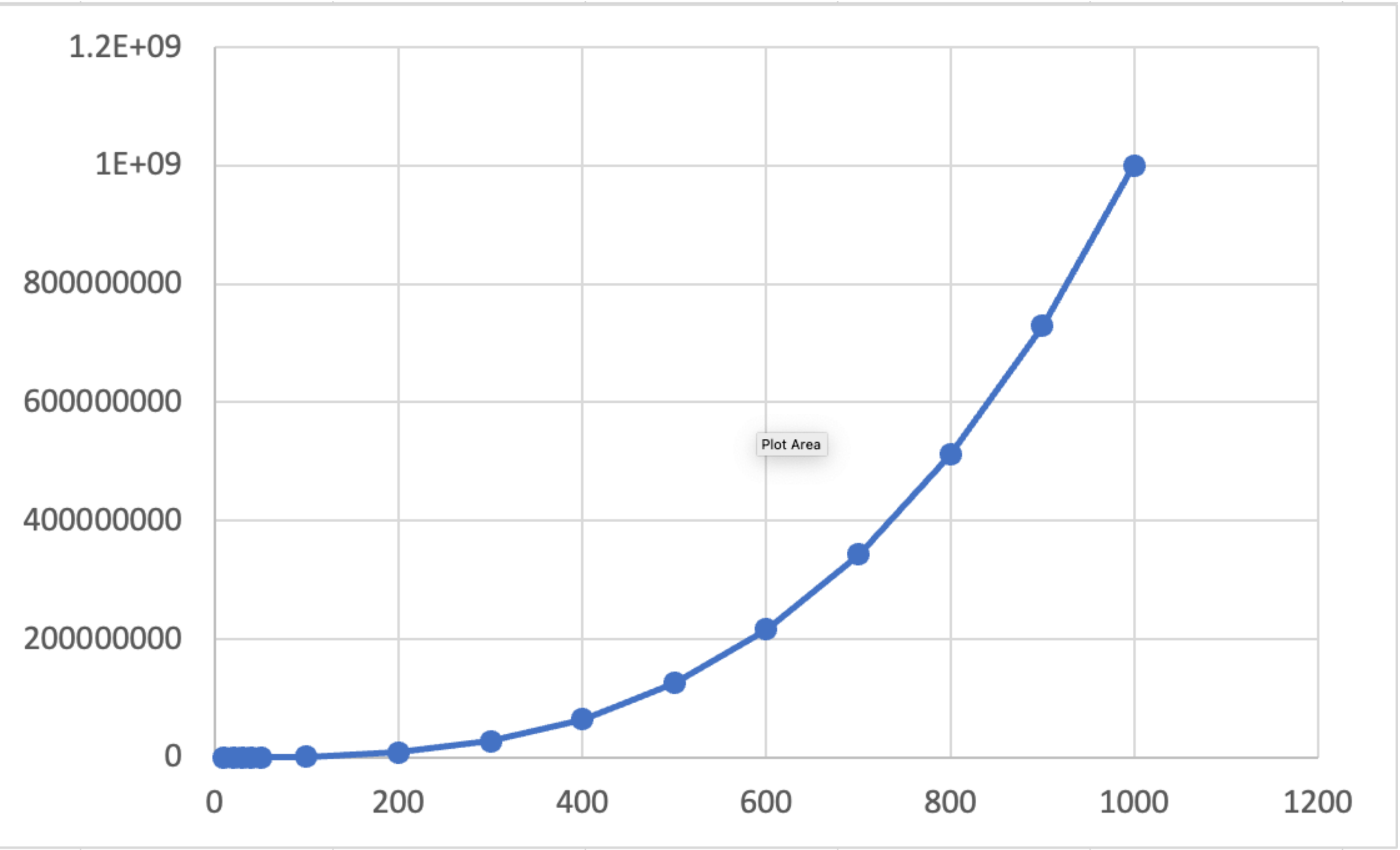
Big O: Quadratic complexity -> O(n^2)



Big O: Cubic complexity -> O(n^3)

```python
def func_cubic(values):
    # example of a cubic complexity function
    # the number of operations is proportional to the cube of
    # the number of items in the list
    sum = 0
    for val in values:
        for val2 in values:
            for val3 in values:
                sum = sum + val + val2 + val3
    return sum
```

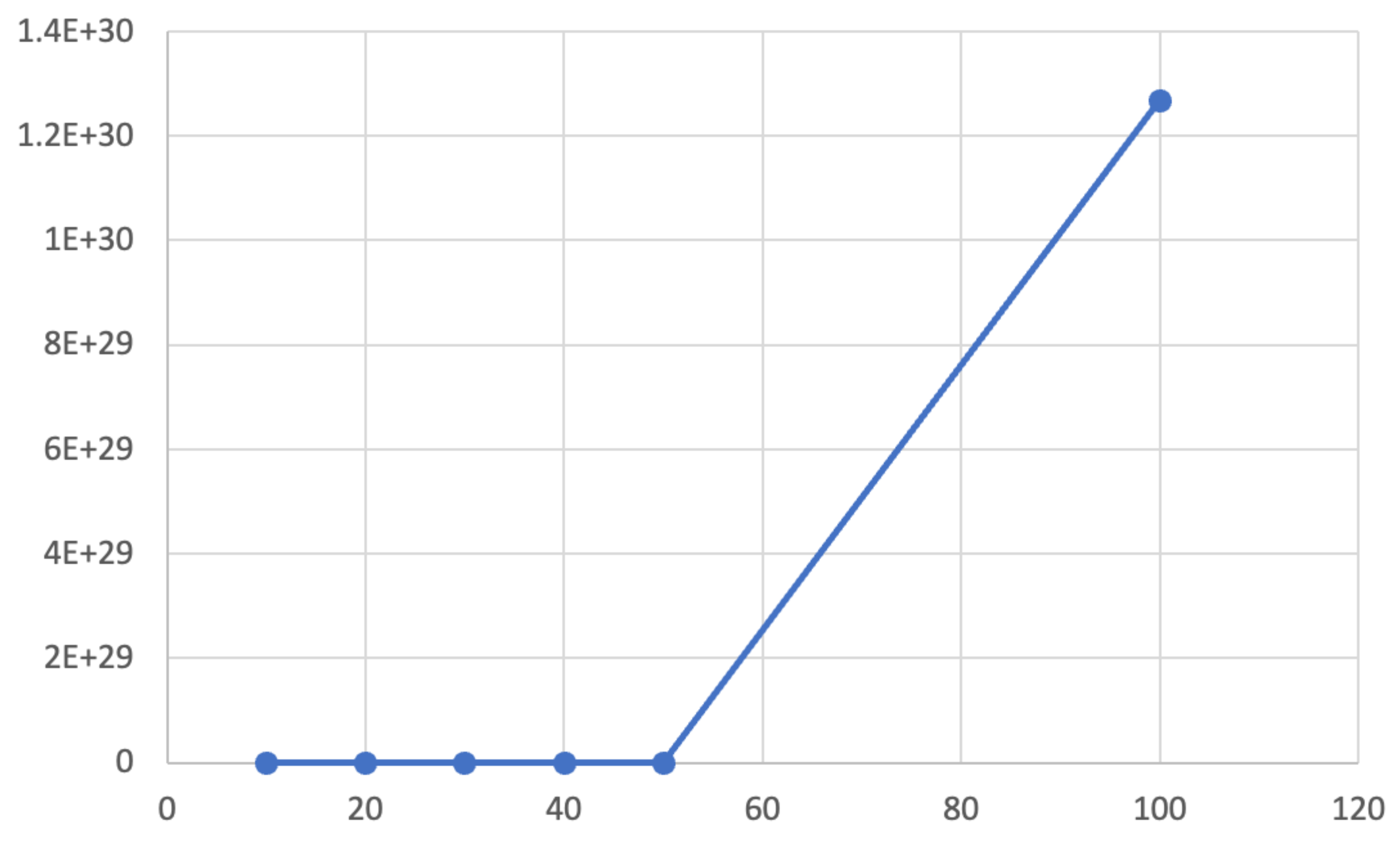
Big O: Cubic complexity -> O(n^3)



Big O: Exponential complexity -> O(2^n)

```python
def func_exponential(v):
    # example of an exponential complexity function
    # the number of operations is proportional to 2^n
    if v <= 1:
        return v
    return func_exponential(v-1) + func_exponential(v-2)
```

Big O: Exponential complexity -> O(n^3)



Big O: Logarithmic complexity -> O(log(n))

```python
def func_logarithmic(n):
    # example of a logarithmic complexity function
    # the number of operations is proportional to the logarithm of
    # the input

    # number of times we can divide n by 2 before we get 1
    c = 0
    while n > 1:
        n = n / 2
        c += 1
    return c
```

Big O: Logarithmic complexity -> O(log(n))