

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 |
|---------------------------------|---|
| <code>point.setx(xcoord)</code> | Sets the x coordinate of point to $xcoord$ |
| <code>point.sety(ycoord)</code> | Sets the y coordinate of point to $ycoord$ |
| <code>point.get()</code> | Returns the x and y coordinates of point as a tuple (x , y) |
| <code>point.move(dx, dy)</code> | Changes the coordinates of point from the current (x , y) to ($x+dx$, $y+dy$) |

OOP: Point Class – Base Definition

 Point.py > ...

```
1  class Point:
2      'Represents a point in 2-D space'
3      def setx(self, x):
4          'Set the x coordinate of the point'
5          self.x = x
6      def sety(self, y):
7          'Set the y coordinate of the point'
8          self.y = y
9      def get(self):
10         'Return a tuple representing the point'
11         return (self.x, self.y)
12     def move(self, dx, dy):
13         'Move the point by dx and dy'
14         self.x += dx
15         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__`

```
Python Point.py > Point
1 class Point:
2     'Represents a point in 2-D space'
3
4     def __init__(self, xcord, ycord) -> None:
5         'Initialize the position of a new point'
6         self.x = xcord
7         self.y = ycord
8         # 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__`

```
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 |
|------------------------------|---------------------------------------|---------------------------------|--------------------|
| <code>x + y</code> | <code>x.__add__(y)</code> | Addition | Concatenation |
| <code>x - y</code> | <code>x.__sub__(y)</code> | Subtraction | — |
| <code>x * y</code> | <code>x.__mul__(y)</code> | Multiplication | Self-concatenation |
| <code>x / y</code> | <code>x.__truediv__(y)</code> | Division | — |
| <code>x // y</code> | <code>x.__floordiv__(y)</code> | Integer division | — |
| <code>x % y</code> | <code>x.__mod__(y)</code> | Modulus | — |
| <code>x == y</code> | <code>x.__eq__(y)</code> | Equal to | |
| <code>x != y</code> | <code>x.__ne__(y)</code> | Unequal to | |
| <code>x > y</code> | <code>x.__gt__(y)</code> | Greater than | |
| <code>x >= y</code> | <code>x.__ge__(y)</code> | Greater than or equal to | |
| <code>x < y</code> | <code>x.__lt__(y)</code> | Less than | |
| <code>x <= y</code> | <code>x.__le__(y)</code> | Less than or equal to | |
| <code>repr(x)</code> | <code>x.__repr__()</code> | Canonical string representation | |
| <code>str(x)</code> | <code>x.__str__()</code> | Informal string representation | |
| <code>len(x)</code> | <code>x.__len__()</code> | — | Collection size |
| <code><type>(x)</code> | <code><type>.__init__(x)</code> | 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 > ...  
1  class Point:  
2      'Represents a point in 2-D space'  
3  
  
26  def __str__(self) -> str:  
27      'Return a string representation of the point'  
28      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 > ...

```
1  class Animal:
2      'represents a generic animal'
3
4      def setSpecies(self, specs):
5          'set the species of the animal'
6          self.species = specs
7      def setLanguage(self, lang):
8          'set the language of the animal'
9          self.language = lang
10     def speak(self):
11         'prints a sentence by the animal'
12         print('I am a ' + self.species + ' and I speak ' + self.language)
13     def getSpecies(self):
14         'returns the species of the animal'
15         return self.species
```

OOP: Inheritance + Overloading


Animals >  Bird.py > ...

```
1  from Animal import Animal
2  class Bird(Animal):
3
4      def speak(self):
5          print('{}!'.format(self.language)*3)
```

Animals >  Dog.py > ...

```
1  from Animal import Animal
2  class Dog(Animal):
3
4      def speak(self):
5          print(self.language)
```

OOP: Inheritance + Overloading

```
Animals >  Main.py > ...  
1  from Animal import Animal  
2  from Bird import Bird  
3  from Dog import Dog  
4  
5  squirrel = Animal()  
6  squirrel.setSpecies('Flying squirrel')  
7  squirrel.setLanguage('Squeak')  
8  print(squirrel.getSpecies())  
9  squirrel.speak()  
10  
11 parrot = Bird()  
12 parrot.setSpecies('Parrot')  
13 parrot.setLanguage('Squawk')  
14 print(parrot.getSpecies())  
15 parrot.speak()  
16  
17 dog = Dog()  
18 dog.setSpecies('Dog')  
19 dog.setLanguage('Woof')  
20 print(dog.getSpecies())  
21 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__

```
1 from EmptyQueueError import EmptyQueueError
2 class Queue:
3     'a classic queue class'
```

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

Queue > EmptyQueueError.py > EmptyQueueError

```
1 class EmptyQueueError(Exception):
2     pass
```

OOP: Custom Errors

Queue >  main.py > ...

```
1  from Queue import Queue
2  # initialize a queue
3  q = Queue()
4  # enqueue some items
5  q.enqueue("apple")
6  q.enqueue("banana")
7  q.enqueue("cherry")
8  # print the queue
9  print(q)
10 # dequeue some items
11 print(q.dequeue())
12 print(q.dequeue())
13 print(q.dequeue())
14 # print the queue (should be empty)
15 print(q)
16 # try to dequeue from an empty queue (should raise an exception)
17 print (q.dequeue())
18
```

```
stefan@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
```

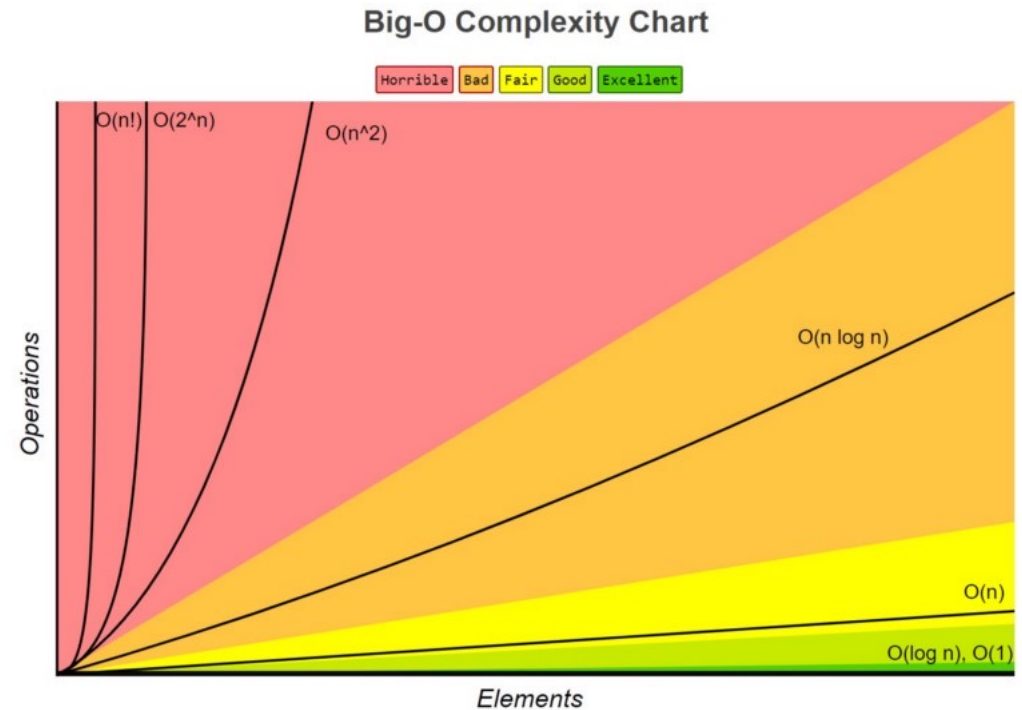
Big O Notation

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 | $O(n \cdot \log^* n)$ |



Graph: [freecodecamp](https://www.freecodecamp.org/algorithm/complexity)

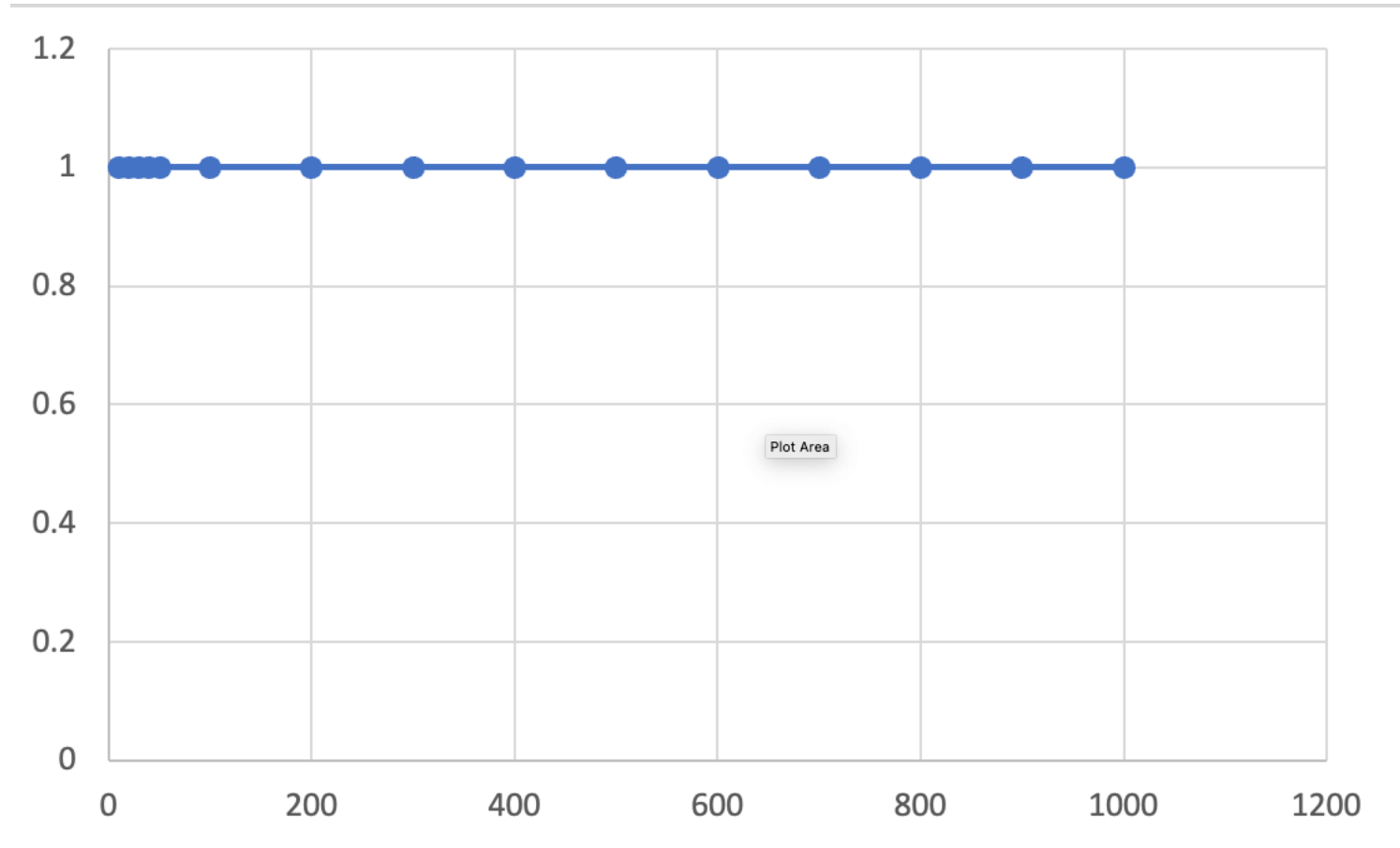
Big O: How do we determine the degree ?

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

Big O: Constant complexity -> O(c)

```
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
```

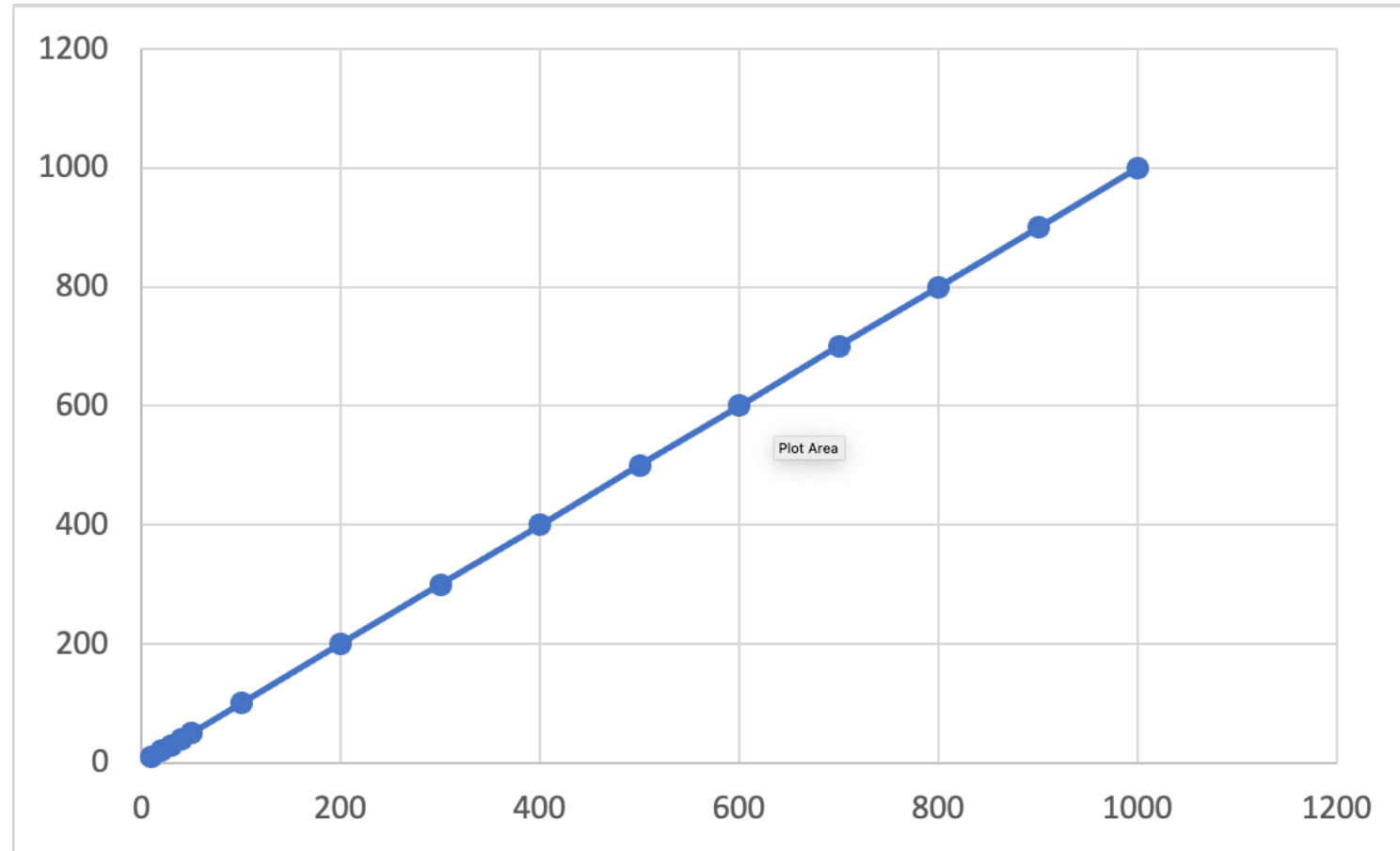
Big O: Constant complexity $\rightarrow O(c)$



Big O: Linear complexity $\rightarrow O(n)$

```
3  ✓ def func_linear(values):  
4      # example of a linear complexity function  
5      # the number of operations is directly proportional  
6      # to the number of items in the list  
7      sum = 0  
8  ✓  for val in values:  
9      |     sum = sum + val  
10     return sum  
11
```

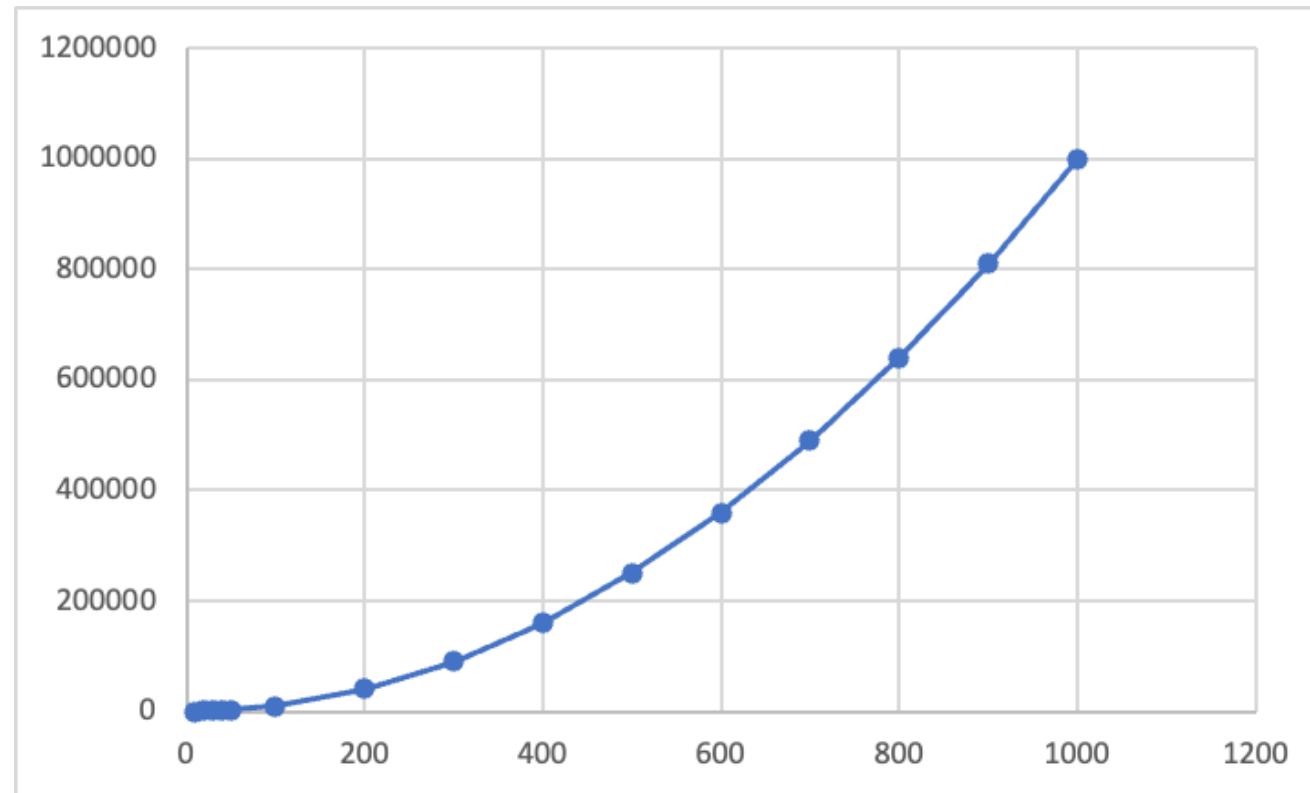
Big O: Linear complexity $\rightarrow O(n)$



Big O: Quadratic complexity $\rightarrow O(n^2)$

```
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
```

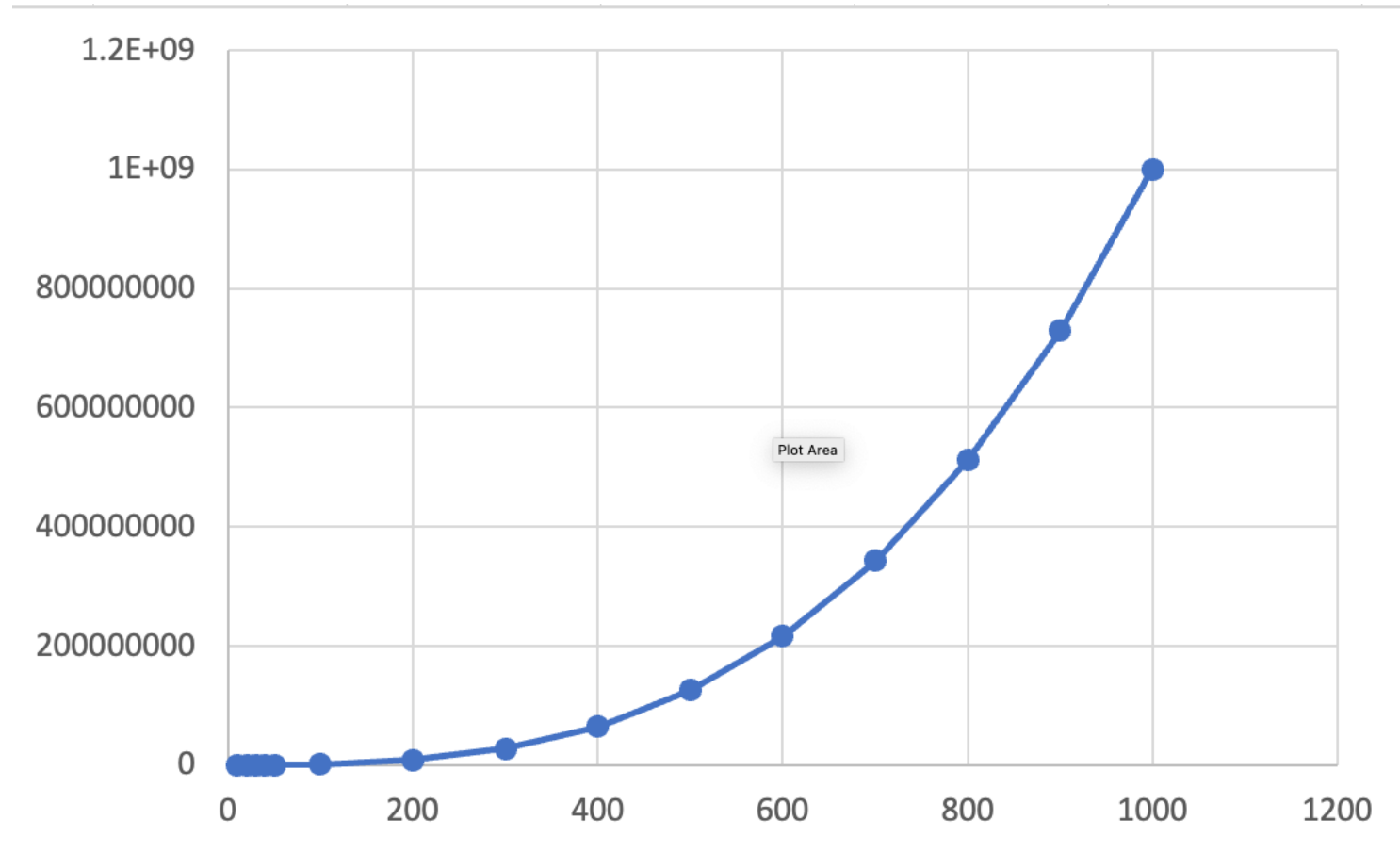

Big O: Quadratic complexity $\rightarrow O(n^2)$



Big O: Quadratic complexity -> $O(n^3)$

```
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
```

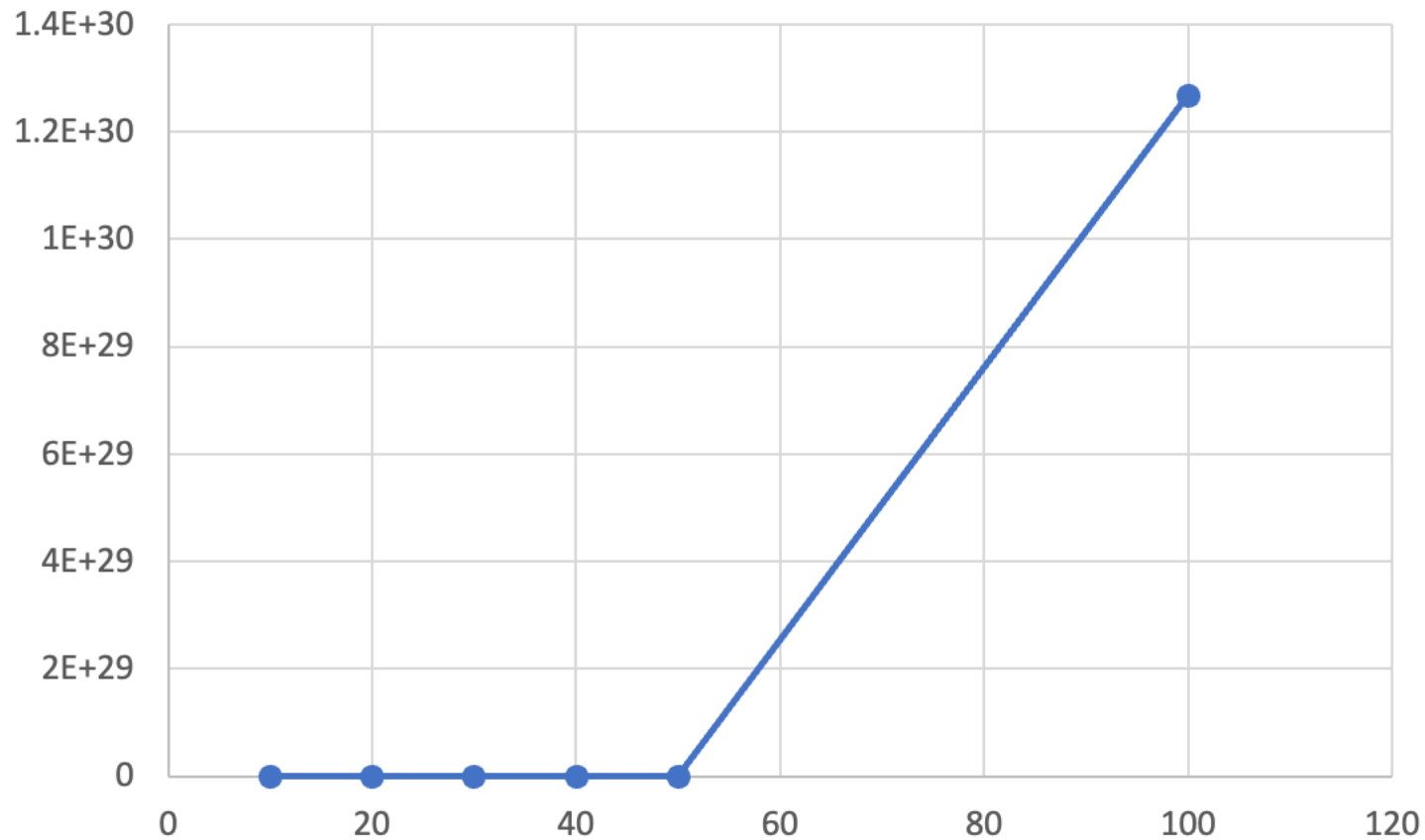
Big O: Quadratic complexity -> $O(n^3)$



Big O: Exponential complexity $\rightarrow O(2^n)$

```
def func_exponential(values):  
    # example of an exponential complexity function  
    # the number of operations is proportional to  $2^n$   
    # where n is 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: Quadratic complexity -> $O(n^3)$



```
x stefan 🐻❤️ code/BigO python3 3_quadratic.py
```

```
List of 10 items
Number of operations: 100
Sum: 1100
List of 20 items
Number of operations: 400
Sum: 8400
List of 30 items
Number of operations: 900
Sum: 27900
List of 40 items
Number of operations: 1600
Sum: 65600
List of 50 items
Number of operations: 2500
Sum: 127500
List of 60 items
Number of operations: 3600
Sum: 219600
List of 70 items
Number of operations: 4900
Sum: 347900
List of 80 items
Number of operations: 6400
Sum: 518400
List of 90 items
Number of operations: 8100
```

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

```
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 $\rightarrow O(\log(n))$

