# Problem – Abstraction - Computable Problem

**Computational problem:**

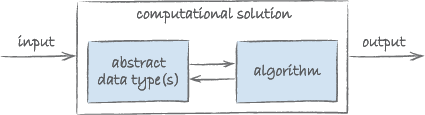
The statement of a computational problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

(Cormen, et al., 2009, p. 6)

**Computable problem** – a problem for which an algorithm can be created which solves the problem for every possible input.

**Abstraction** – simplification:

* abstraction as modelling of a part of reality. Uses data structures and abstract data types to model the data relevant to the problem. Algorithm will use data structures to transform input into output.



* abstraction as encapsulation, where the inner mechanisms of a model are hidden from users.

**Abstract Data Type (ADT)** is a description of a data structure purely in terms of the operations that can be carried out on it. The ADT offers *no* account of how these operations are programmed, how the data is stored, the programming language used, or any other specific details.

**Data Structure** – implemented (written using particular algorithms and programming language) ADT

# Developing Algorithm

Using example of algorithm to find highest number in a sequence.

### Getting inputs and outputs

|  |  |
| --- | --- |
| **Name:** | FindHighest |
| **Inputs:** | A sequence of integers *I* = (*i*1, *i*2, *i*3, …, *in*) |
| **Outputs:** | An integer *h* |

### Initial inside

A vague and rather abstract idea of the steps necessary:

Set highest to first item in intSeq. Examine each number in intSeq, one by one. If the current number is larger than highest, then set highest to the current number.

### Expressing algorithm

As you’ve learned, algorithms are simply mixtures of iterations, sequences and selections. The task now is to extract the right combination of these from the initial insight, and express them in a clear form, and with the right level of detail.

The language of the initial insight will generally supply the clues required to outline the algorithm. Expressions like ‘examine each name’ and ‘one by one’ imply an iteration; ‘if’ and ‘then’ suggest selections. Names like ‘item’ and ‘mark’ may refer to variables in which results and intermediate calculations are stored.

initialise **highest** to the first element in **intSeq**

**ITERATE** over each **item** in **intSeq**

**IF** **item** is greater than **highest**

set **highest** to **item**

return **highest**

# What is Algorithm?

* a clearly specified set of simple instructions to be followed to solve a problem
  + takes a set of values, as input and
  + produces a value, or set of values, as output

## Why need algorithm analysis?

* Writing a working program is not good enough
* The program may be inefficient!
* If the program is run on a **large data set**, then the running time becomes an issue

Example **algorithm**:

Given a list of N numbers, determine the kth largest, where k <= N.

Algorithm 1:

* Read N numbers into an array
* Sort the array in decreasing order by some simple algorithm
* Return the element in position k

[Python:](../../../../Documents/training/AlgorithmsAndDataStructures/PythonExamples/largestNumOfGivenIndex.py)

#!/usr/bin/env python3

#Input

#Create a list of numbers – [] – means list in Python

numbers = [2,7,4,6,9,5,1]

k = 1

def largesNumOfGivenIndex(numbers, index):

numbers.sort()

numbers.reverse()

print(numbers)

print("Larges munber of index " +str(index)+ " is: " + str(numbers[index]))

#Run the function defined above

largesNumOfGivenIndex(numbers, k)

## We only analyse correct algorithms

### An algorithm is correct

* If, for every input instance, it halts with the correct output

### Incorrect algorithms

* Might not halt at all on some input instances (infinite loop)
* Might halt with other than the desired answer

### Analysing an algorithm

* Predicting the resources that the algorithm requires
* Resources include
  + Memory
  + Communication bandwidth
  + Computational time (usually most important)

### Factors affecting the running time

* computer (slower / faster)
* implementation language / compiler (each language has its characteristics and may cause the algorithm to run slower or faster)
* algorithm used (each problem has multiple solutions which may be better (more efficient ) than the others)
* input to the algorithm
  + The content of the input affects the running time (sorted / unsorted)
  + typically, the input size (number of items in the input) is the main consideration
    - E.g. sorting problem  the number of items to be sorted
    - E.g. multiply two matrices together  the total number of elements in the two matrices
* Machine model assumed
  + Instructions are executed one after another.

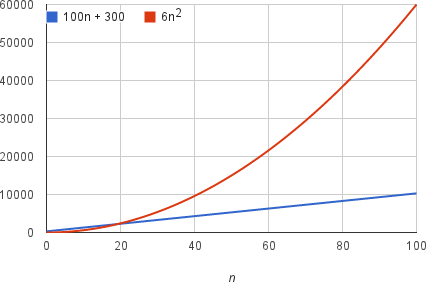
### Measuring running time of algorithm

**Size of input** is independent of hardware or software used to implement and run algorithm therefore it is better to compare algorithms on their own, eliminating the side effects or compilations of hardware and software used. Analysing the algorithm itself in separation of other factors.

We can think about the running time of the algorithm as a function of the size of its input. (Less input to search function (less items to search in) will make search function run faster).

**Rate of growth of running time** – how fast the function grows with the input size. How fast the running time of algorithm grows with input size.

For example, suppose that an algorithm, running on an input of size *n*, takes 6n2 + 100n +300 machine instructions. The 6n2 term becomes larger than the remaining terms, once *n* becomes large enough, 20  in this case. Here's a chart showing values of 6n2 and 100n + 300  for values of *n* from 0 to 100:



By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—without getting mired in details that complicate our understanding. When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**. We'll see three forms of it: big-Θ (Theta) notation, big-O (order of magnitude) notation, and big-Ω (Omega) notation.

## Worst- big-O / average- big-Θ / best-case- big- Ω

### Worst-case running time of an algorithm

* The longest running time for any input of size n
* An upper bound on the running time for any input  guarantee that the algorithm will never take longer
* Example: Sort a set of numbers in increasing order; and the data is in decreasing order
* The worst case can occur fairly often
  + E.g. in searching a database for a particular piece of information

### Best-case running time

* sort a set of numbers in increasing order; and the data is already in increasing order

### Average-case running time

* May be difficult to define what “average” means

## Running-time of algorithms

* **Big O** (**order of magnitude**) notation is often used to describe how the size of the input data affects an algorithm's usage of computational resources (usually running time or memory).
* The symbol O is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, usually simpler function.

## Big-O: example

### Let f(N) = 2N2. Then

* f(N) = O(N4)
* f(N) = O(N3)
* f(N) = O(N2) (best answer, asymptotically tight)
* Big-O notation (O(f(n)) = O(N2): reads “order N-squared” or “Big-Oh N-squared”

### More examples:

* N2 / 2 – 3N = O(N2)
* 1 + 4N = O(N)
* 7N2 + 10N + 3 = O(N2)
* log N + N = O(N)

### Growth Rate – from smallest to greatest

Here's a list of functions in asymptotic notation that we often encounter when analysing algorithms, listed from slowest to fastest growing. This list is not exhaustive; there are many algorithms whose running times do not appear here:

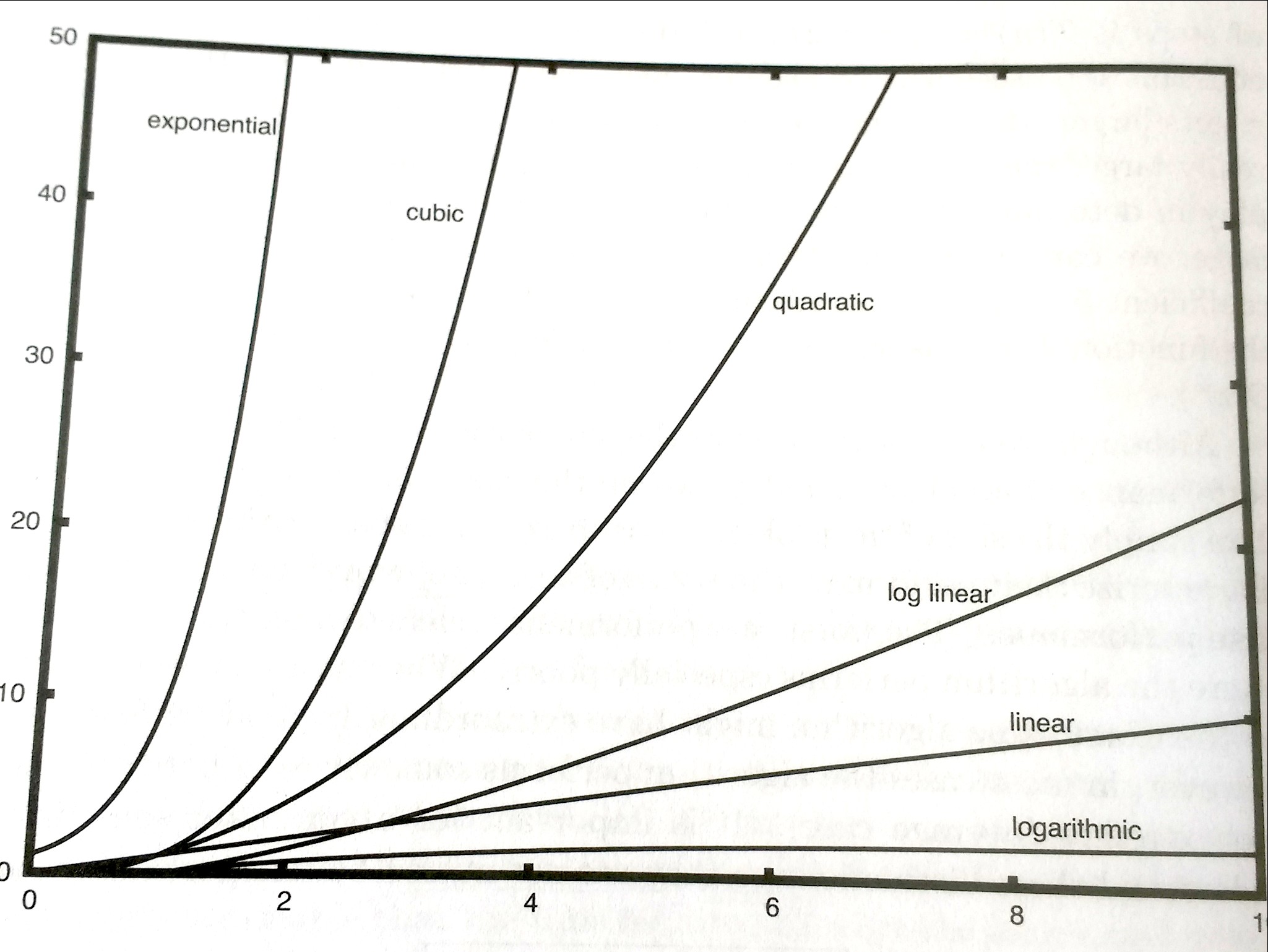
Θ(1) – constant time - always the same no matter how big the input size; ex.: indexing Python list: list[x]

Θ(log n) – logarithmic function; ex.: binary search – halves the input size with each iteration.

Θ(n) – linear – iterating a list, it will need to iterate *n* times for list of size *n*

Θ(n lg n) – Log Linear

Θ(n2) – square/quadratic function; ex.: two nested loop – for loop nested inside another for loop, iterate *n* \* *n* times

 Θ(n3) – cubic; ex.: 3 nested loops

Θ(2n) – exponential

### Doubling the input size

* f(N) = constant --------> f(2N) = f(N) = c
* f(N) = log N -----------> f(2N) = f(N) + log 2
* f(N) = N ---------------> f(2N) = 2 f(N)
* f(N) = N2 --------------> f(2N) = 4 f(N)
* f(N) = N3 --------------> f(2N) = 8 f(N)
* f(N) = 2N --------------> f(2N) = f2(N)

### Advantages of algorithm analysis

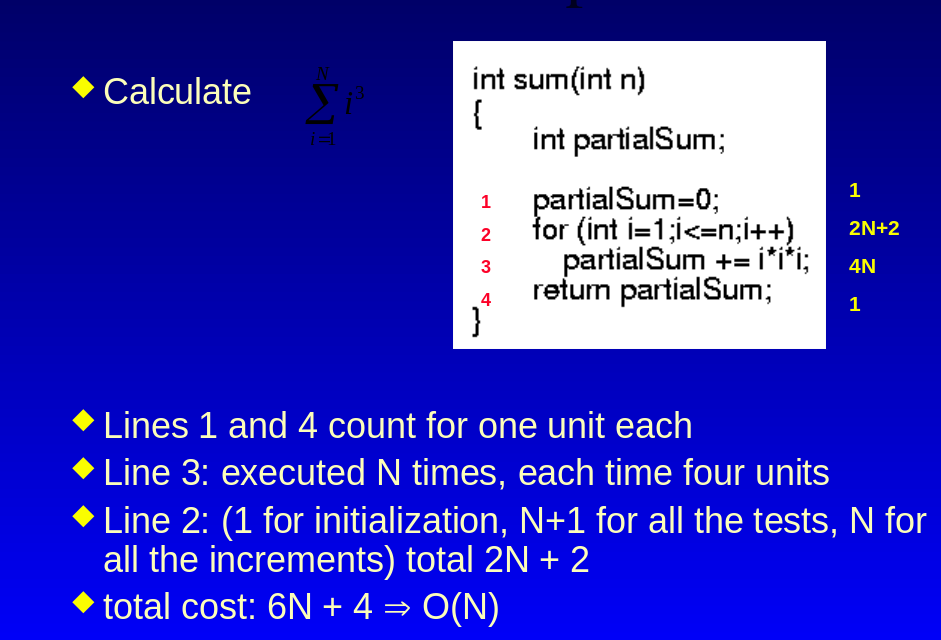
* To eliminate bad algorithms early
* pinpoints the bottlenecks, which are worth coding carefully

### Big-O algorithm analysis – input size to execution time relation

Analyse algorithm efficiency in terms of execution time, independent of any particular program or computer. Defining the relation between input size and execution time, how increasing input size affects execution time.

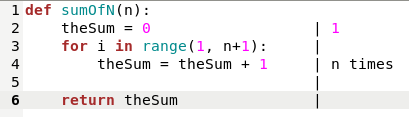
Basic unit of operation - We need to quantify the number of operations or steps in the algorithm. If each of this steps is considered as basic unit of computation then the execution time of the algorithm can be expressed as the number of steps required to solve the problem.

### Example 1:

* Lines 1 and 4 count one unit each (assignment and return statement) → 1 +1
* Line 3: is executed N times, each time 4 units – we have =,+,\*,\* executed N times –-> 4N
* Line 2: 1 for initialization of i (int i=1), N+1 for all tests (i<=n – needs to be executed one more time than N to determine when to stop), N times for all increments (i++) --> which gives: 1 + N + 1+ N = 2N + 2
* Total cost: T(N) = 2 + 4N + 2N + 2 = 6N + 4

Above example is counting every operation in the algorithm. In Big-O analysis it is common to select basic unit of operation to simplify the process, after all we are simplifying the problem to understand the dominant term which will grow the most with increased input. Usually a good choice for basic unit of operation is the assignment statement.

### Example 2:



In this example we only count assignment statement and as well we can omit for loop definition (line 3) and return statement (line 6) because it is more important to focus on what steps or calculations are done inside the loop to estimate the size of the problem. In this summation algorithm we can say that 100,000 summation is bigger instance of the problem than 1000 because the accumulation on line 4 will be executed 100,000 times in first case and 1000 times in second.

In this case we have:

* Line 2 – 1 assignment
* Line 4 – n assignments
* Total: T(n) = 1 + n

For Big-O analysis it is not so important to count exact number of operation but to determine the most dominant part which will grow the fastest when input increases. This dominant term will overpower other terms with increasing input.

For example 1:

* where T(n) = 6N + 4
* Big-O will be O(n) - (we use notation of O(T(n)) = N or short version O(N)) because the constant 4 and coefficient 6 will become insignificant with large N

For example 2:

* where T(n) = 1 + n
* O(T(n)) = n or O(n) because constant 1 will become insignificant for Big-O estimation with large input n.

**Any function in Big-O notation can be reduced to its dominant term:**

**- drop lower order terms**

**- drop constant multipliers**

Any function of following structure: *Xn +Y (like 5n+10)* where X,Y are constants will be O(n)

Any function of following structure: *Xn2 +Yn + Z (like 5n2+100n+1000)* where X,Y,Z are constants will be O(n2)

Any function of following structure *Xn3 +Yn2 + Zn + X* where X,Y,Z are constants will be O(n3)

- - || - - Xn4 +Yn2 + Zn - - || - - O(n4)

## General Rules

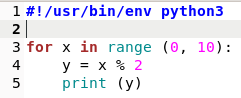
It is not necessary to count exact number of statements and come up with mathematical representation of program time complexity to make correct Big-O analysis. Here are rules which can be used to quickly asses computer programs.

### Simple statements - constant

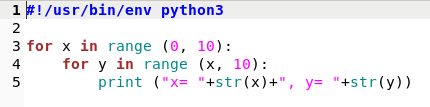
If we have only simple statements (no loops etc…) Big-O will always be constant, it does not change with increasing size of input. O(1) - constant

### For loops

* **Single loop – O(n)**

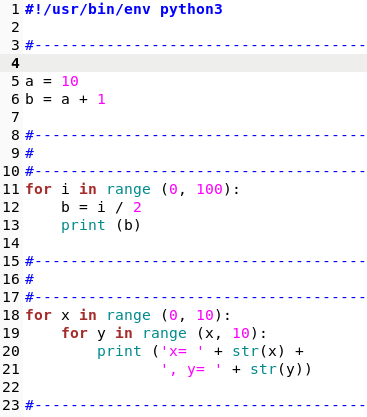
Single loop if it contains only simple statements (does not matter how many) will always be of linear growth rate – O(n). The number of loops executed is proportional to the size of input.

* **Nested loops - O(nx) – where x - is the number of nested loops**

 Nested loops like the one below will always evaluate to O(n2). Even that the inner loop like in this example does not iterate through full range of n. In this example it decreases n by 1 each time it runs. But this factor becomes a coefficient which becomes insignificant with large size of n. Even if we had 12n2 the Big-O will still be O(n2) because 12 is a coefficient which we can drop in Big-O approximation.

If we have more nested loops like 3 of them than proportionally Big-O will be O(n3) The power will be the number of nested loops.

### Consecutive statements

In this example we have 3 parts separate with blue dashed lines.

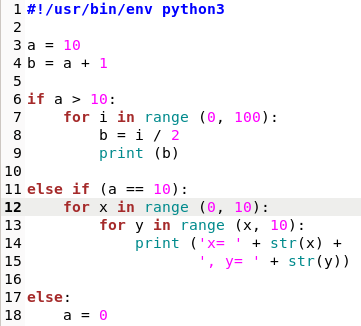
First part consists of simple statements which will be of constant time complexity O(1)

Second is a single loop which will be O(n)

Third part is a nested loop which will evaluate to O(n2)

In case of consecutive statement we need to separate logical groups like above which have same Big-O value and add them together. In this case it will be: O(1) + O(n) + O(n2) . In this case the dominant factor is O(n2) because it will grow the fastest of all factors and whole program will evaluate to O(n2) because other terms become insignificant with large input size.

### If else / switch statements

With if else statements where only one of the branches will be executed since they are exclusive we need to find the one which has the highest Big-O. In Big-O analysis we are looking for the worst case scenario.

In this example we have 4 parts:

Simple statement on limes 3 and 4 executed once which will be O(1) – constant.

Then we have if else statement with 3 branches:

* First branch has single loop which is O(n) – linear
* Second branch is a nested loop which is O(n2)
* Third branch is constant O(1) because it has only a simple statement.

So we have a simple statements on the beginning plus the the worst case from if else statement which results in: O(1) + O(n2) = O(n2)