Complexity and Big-O

Can computer compute everything???
Why and Why not?

The general answer is NO, computer can't compute everything!
We will discuss reasons in this section.

Let's try out the program we know

- Run tower_of_hanoi with 10, 15, 20, 23, 24, or 30 discs …

The Wheat and Chessboard Problem

- The inventor of chess (in some tellings Sessa, an ancient Indian Minister) request his ruler give him wheat according to the Wheat and Chessboard Problem. The ruler laughs it off as a meager prize for a brilliant invention, only to have court treasurers report the unexpectedly huge number of wheat grains would outstrip the ruler's resources.

https://en.wikipedia.org/wiki/Wheat_and_chessboard_problem

The problem

1	2	4	8	16	32	…	
							???

Add them up 1+2+4+… = 1+2+4+…+2^{64} = 18,446,744,073,709,551,615 !!!

What does it have anything to do with CS?

If your program requires 2^n steps to compute something, it is not practical to expect any results in a meaningful time frame for the problem of reasonable large input size n, e.g., a few tens or a few hundreds.

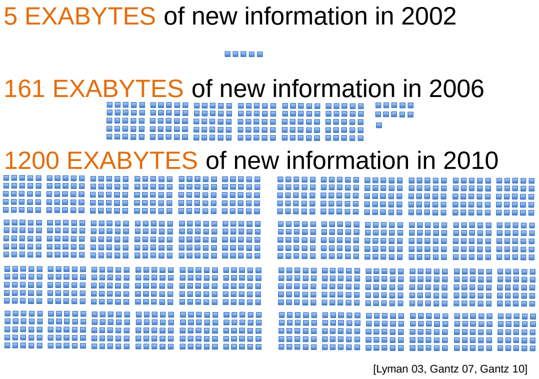
In our Wheat and Chessboard problem, n = 64.
Steps = 18,446,744,073,709,551,615.
A modern desk-top computer (4GHz) can do roughly 4 billion steps per second. To go through that many steps, it would take

584,943,368 centuries!!!

Why does this matter?

Computers are so fast! But…

- Large Scale Data
  - Google, Twitter, Facebook.. Big Data



5 EXABYTES of new information in 2002

161 EXABYTES of new information in 2006

1200 EXABYTES of new information in 2010

[Lyman 03, Gantz 07, Gantz 10]

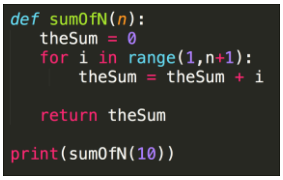
2019????

Why does this matter?

Computers are so fast! But…

- Large Scale Data
  - Google, Twitter, Facebook.. Big Data
- Limited Resources
  - phones, watches, wearable computing
- High Performance Environments
  - milliseconds matter

How do we know what matters in code?



```
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    return theSum

print(sumOfN(10))
```

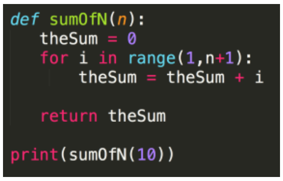
1

1 * (n)

1

n+2

How do we know what matters in code?

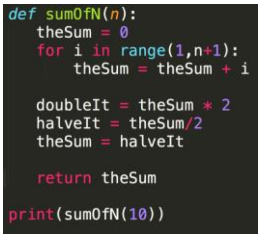


```
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    return theSum

print(sumOfN(10))
```

n+2



```
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    doubleIt = theSum * 2
    halveIt = theSum/2
    theSum = halveIt

    return theSum

print(sumOfN(10))
```

n+5

O(n)

Pay attention to what changes as the variables increases

Selection sort

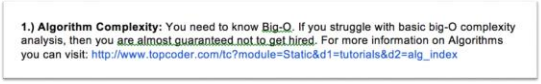
```
def sortByTitle(self):
    '''
    Sort the album by the title, without change the original data
    Here we use insert sort as students saw it in hw3
    '''
    for i in range(len(self)):                                   # n
        minIndex = self.findMinIndex(self.sortedByTitles, i)     # n-i
        self.swap(minIndex, i, self.sortedByTitles)              # 1
```

$$steps = \sum_{i=1}^{n} n - i = n^2 - \frac{n(n+1)}{2} = n^2 - \frac{n^2}{2} - \frac{n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

O(n^2)

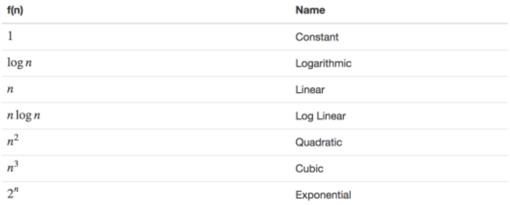
Big-O Notation

- No need to count precise number of steps
- Classify algorithms by order of magnitude
  - Execution time
  - Space requirements
- Big O gives us a rough **upper bound**
- Goal is to give you intuition



1.) Algorithm Complexity: You need to know Big-O. If you struggle with basic big-O complexity analysis, then you are almost guaranteed not to get hired. For more information on Algorithms you can visit: http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=alg_index

Describing Growth

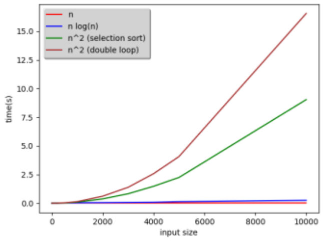


f(n)	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

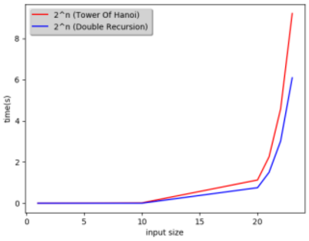
The first one is constant time O(1), the second one is logarithm time O(log n), the last one is exponential time $O(2^n)$, rest polynomial time $O(n^k)$.

Let's try the program … bigO.py

For polynomial time



For exponential time



Pay attention to the problem size and the actual timing.