



SF STATE

SAN FRANCISCO STATE UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

The Efficiency of Algorithms Computing Complexity

Duc Ta

Choosing an implementation = **balance**(efficiency, frequency)

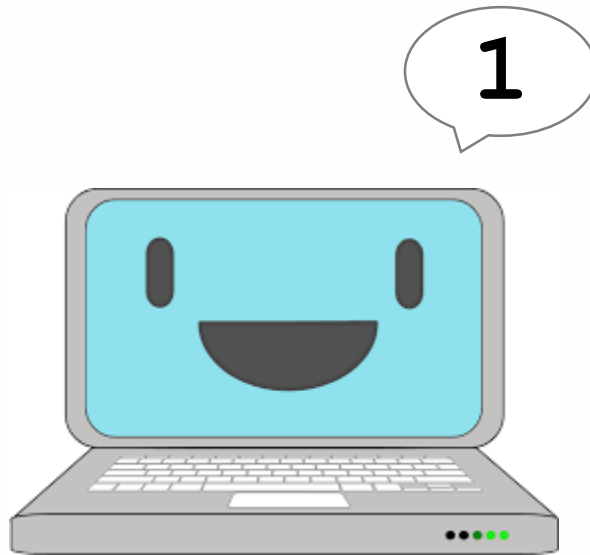
- Look for **significant differences** in **efficiency**
 - **The use of time**
 - **The use of space** (memory)
- **Frequency of operations**
 - Consider how frequently particular ADT operation occur in a given application.
 - Sometimes seldom-used but critical operations must be efficient.
- **Best-case Analysis**
 - Determine **minimum** amount of **time** an algorithm requires to solve problems of size n.
- **Worst-case analysis**
 - Determine **maximum** amount of **time** an algorithm requires to solve problems of size n.
- **Average-case analysis**
 - Determine **average** amount of **time** an algorithm requires to solve problems of size n.

THE EFFICIENCY OF ALGORITHMS

(AN INTRODUCTION)

COUNTING OPERATIONS

COUNTING OPERATIONS



1 time unit per operation
assignment
comparison
addition
...
return
...

COUNTING OPERATIONS

```
sum (x, y){  
    return x + y  
}
```

Total time: 2

COUNTING OPERATIONS

```
sumOfList(L, n) {  
    total = 0;  
  
    for (int i = 0; i < n; i++) {  
  
        total = total + L[i];  
    }  
    return total  
  
}
```

COUNTING OPERATIONS

```
sumOfList(L, n) {  
    total = 0;  
    for (int i = 0; i < n; i++) {  
        total = total + L[i];  
    }  
    return total;  
}
```

1

$$1 + 1 \cdot (n + 1) + 2 \cdot n = 3n + 2$$

$$(1 + 1) \cdot n = 2n$$

1

Total time: $5n + 4$

COUNTING OPERATIONS

```
sumOfList(L, n) {  
    total = 0;  
    for (int i = 0; i < n; i++) {  
        total = total + L[i];  
    }  
    return total;  
}
```

1

$$1 + 1*(n + 1) + 2*n = 3n + 2$$

$$(1 + 1) * n = 2n$$

1

Total time: $5n + 4$

Measuring an Algorithm's Efficiency

- “What important, however, is **not the exact count of operations**, but the **general behavior** of the algorithm.”
- The function $5n + 4$ is directly proportional to n . Do not count every operation. Count the significant contributors. → File Manger: [BigO-GrowthRates.xlsx](#)
- An algorithm's **basic operation** is the most significant contributor to its total time requirement.

COUNTING BASIC OPERATIONS

Basic Operation

- **Basic operation** is the **most significant contributor** to an algorithm's **total time requirement**.
 - If an algorithm that sees whether an array contains a particular object then **comparison** is its **basic operation**.
 - If an algorithm that adds a group of numbers then **addition** is its **basic operation**.
 - → Count the “**action**” statements. Statements directly related to accomplishing goal: Additions, Multiplications, Comparisons, Moves.

Basic Operation

- The **most frequent** operation is **not** necessarily the basic operation because they do not affect the final conclusion about algorithm speed.
 - Assignments are most frequent but rarely are basic.
 - Same as initializations of variables or operations which control loops
 - Ignore “**bookkeeping**” statements like displaying statements.

Basic Operation

- Which one?

$$\sum_{k=1}^n k = 1 + 2 + 3 + \dots + n$$

Algorithm A	Algorithm B	Algorithm C
<pre>sum = 0 for i = 1 to n sum = sum + i</pre>	<pre>sum = 0 for i = 1 to n { for j = 1 to i sum = sum + 1 }</pre>	<pre>sum = n * (n + 1) / 2</pre>

Basic Operation




- Which one?

$$\sum_{k=1}^n k = 1 + 2 + 3 + \dots + n$$

Algorithm A	Algorithm B	Algorithm C
<pre>sum = 0 for i = 1 to n sum = sum + i</pre>	<pre>sum = 0 for i = 1 to n { for j = 1 to i sum = sum + 1 }</pre>	<pre>sum = n * (n + 1) / 2</pre>

Basic Operation

- Which one?

Algorithm A	Algorithm B	Algorithm C
<pre>sum = 0 for i = 1 to n sum = sum + i</pre> 	<pre>sum = 0 for i = 1 to n { for j = 1 to i sum = sum + 1 }</pre> 	<pre>sum = n * (n + 1) / 2</pre> 

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n + 1) / 2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2 + n) / 2$	3

THE EFFICIENCY OF ALGORITHMS

MORE PRACTICE PROBLEM

Count Operations:

- Please count the number of operations (not only the basic operations) in the code block below:

```
int i, n = 10, sum = 0;  
for (i = 0; i <= 2n, i++) {  
    sum += i + 3;  
}
```

BIG O

Big-Oh Notation

$$f() = 5n^3 + 20n^2 + 19$$

- The lower order terms become increasingly less relevant as n increases
- So we say that the algorithm is order n^3 , which is written $O(n^3)$
- This is called Big-Oh notation
- There are various Big-Oh categories
- Two algorithms in the same category are generally considered to have the same efficiency, but that does not mean they have equal growth functions or behave exactly the same for all values of n .

Growth Function	Order	Label
$t(n) = 17$	$O(1)$	constant
$t(n) = 3\log n$	$O(\log n)$	logarithmic
$t(n) = 20n - 4$	$O(n)$	linear
$t(n) = 12n \log n + 100n$	$O(n \log n)$	$n \log n$
$t(n) = 3n^2 + 5n - 2$	$O(n^2)$	quadratic
$t(n) = 8n^3 + 3n^2$	$O(n^3)$	cubic
$t(n) = 2^n + 18n^2 + 3n$	$O(2^n)$	exponential

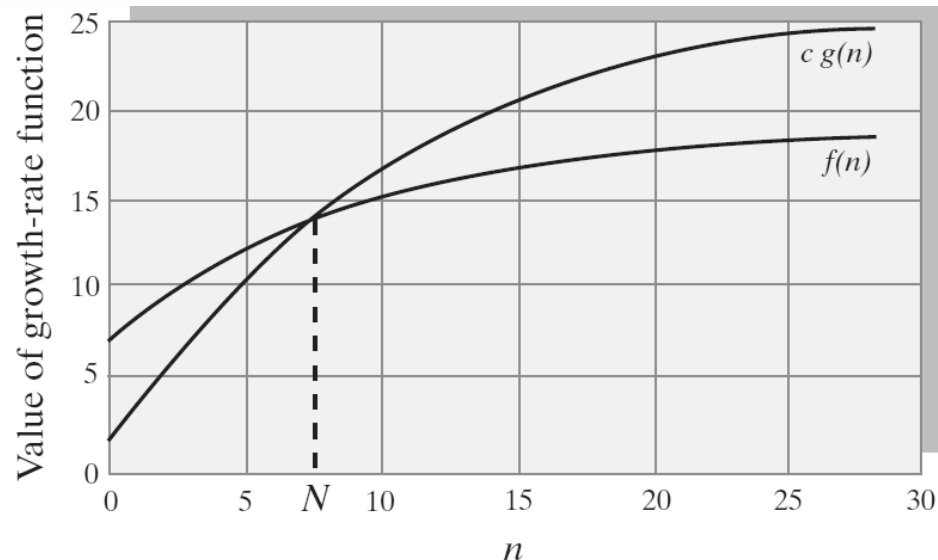
Big-Oh Notation, time complexity, formal definition

Formal definition of Big-Oh

A function $f(n)$ is of order at most $g(n)$ – that is, $f(n)$ is $O(g(n))$ – if:

A positive real number c and positive integer N exist such that $f(n) \leq c * g(n)$ for all $n \geq N$. That is $c * g(n)$ is an upper bound on $f(n)$ when n is sufficiently large.

In simple terms, $f(n)$ is $O(g(n))$ means that $c * g(n)$ provides an upper bound of on $f(n)$'s growth rate when n is large enough. For all data sets of a sufficient size, the algorithm will always require fewer than $c * g(n)$ basic operations.



Big-Oh Notation

- **$O(1)$, constant:** An algorithm that executes in the **same amount of time** regardless of the amount of data.
- **$O(\log n)$, logarithmic:** Occurs when the **data being used is decreased by roughly 50% each time** through the algorithm.
 - For example, the Binary search. Pretty fast because the $\log(n)$ increases at a dramatically slower rate as n increases.
 - $O(\log n)$ algorithms are very efficient because increasing the amount of data has little effect at some point early on because the amount of data is halved on each run through.
- **$O(n)$, linear:** An algorithm whose time to complete will grow in **direct proportion to the amount of data**.

For example, the linear search. To find all values that match, we will have to look in each and every item in the array.

 - If we just wanted to find one match the Big O is the same because it describes the worst case scenario in which the whole array must be searched.

Big-Oh Notation

- **$O(n \log n)$, $n \log n$:** Most sorts are at least $O(n)$ because every element must be looked at, at least once. The bubble sort is $O(n^2)$ to figure out the number of comparisons we need to make with the Quick Sort we first know that it is comparing and moving values very efficiently without shifting. That means values are compared only once. So each comparison will reduce the possible final sorted lists in half.
 - Comparisons = $\log n!$ (Factorial of n)
 - Comparisons = $\log n + \log(n-1) + \dots + \log(1)$
 - This evaluates to $n \log n$.
- **$O(n^2)$, quadratic:** Time to complete will be **proportional to the square of the amount of data** (Bubble Sort). Algorithms with more nested iterations will result in $O(n^3)$, $O(n^4)$, etc. performance. Each pass through the outer loop $O(n)$ requires us to go through the entire list again so n multiplies against itself or it is squared.

BIG-O, LITTLE-O, OMEGA, THETA

Big-Oh Notation, time complexity

$$1 < \log n < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n < \dots < n^n$$

Big-Oh Notation, time complexity

$$1 < \log n < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n < \dots < n^n$$

$$f(n) = 2n^2 + 3n + 2$$

- **Highest degree** is n^2
- The **Upper Bound** of $f(n)$ is $n^2 < n^3 < \dots < 2^n < 3^n < \dots < n^n$
- The **Lower Bound** of $f(n)$ is $1 < \log n < n < n \log n < n^2$
- The **Tight Bound** is n^2
- We can say $f(n) = O(n^2)$
- Since the big O means the Upper Bound, we can say:
 - $f(n) = O(n^3)$
 - ...
 - $f(n) = O(3^n)$
 - ...
 - $f(n) = O(n^n)$

Big-Oh Notation, time complexity

$$1 < \log n < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n < \dots < n^n$$



Lower Bound, Omega

- $f(n) = \Omega(n^2)$
- ...
- $f(n) = \Omega(\log)$
- ...
- $f(n) = \Omega(1)$

Upper Bound, Big-Oh

- $f(n) = O(n^3)$
- ...
- $f(n) = O(3^n)$
- ...
- $f(n) = O(n^n)$

Tight Bound

If $f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$ then $f(n) = \Theta(n^2)$ Theta
 If $f(n) = O(n^2)$ and $f(n) \neq \Theta(n^2)$ then $f(n) = o(n^2)$ Little-Oh

THE EFFICIENCY OF ALGORITHMS

Algorithm's Efficiency vs. Algorithm's Complexity

- An algorithm's **COMPLEXITY** has both
 - *Time requirements*
 - *Space requirements*
- An algorithm's **EFFICIENCY** means
 - *Its use of time*
 - *Its use of space (memory)*

Algorithm's Efficiency vs. Algorithm's Complexity

- An algorithm's **COMPLEXITY** has both
 - *Time requirements*
 - *Space requirements*
- An algorithm's **EFFICIENCY** means
 - *Its use of time*
 - *Its use of space (memory)*
- Time Complexity: The time it takes to execute
- Space Complexity: The memory it needs to execute
- The **BEST** algorithm (Fastest? Least memory? Your boss likes it?)
 - Usually the “best” solution to a problem balances various criteria such as time, space, generality, programming effort, and so on.

Algorithm Analysis

- **Analysis of Algorithm:** The process of **measuring** the **complexity** of algorithm
- **Measure**
 - **Not** how involved/difficult it is
 - **Not** space/memory complexity
 - But the algorithm's **time complexity**

Algorithm Analysis

- **Analysis of Algorithm:** The process of **measuring** the **complexity** of algorithm
- **Measure**
 - **Not** how involved/difficult it is
 - **Not** space/memory complexity
 - But the algorithm's **time complexity**
- **Problem Size**
 - The number of items that an algorithm processes.
 - E.g. Searching a collection of data → Number of items in the collection
- **Growth Rates**
 - Algorithm Analysis is all about understanding **growth rates**. That is as the amount of data gets bigger (the problem gets bigger), how much more resource will the algorithm require?
 - Use graphs to visualize growth rates

GROWTH RATES

(A REFERENCE)

Typical growth-rate functions are simple because the effect of an inefficient algorithm is **not noticeable** when the problem is **small** → focus on **large problem**. Thus, we only care about **large values** of n when comparing algorithms. We can consider only the dominant term in each growth-rate function.

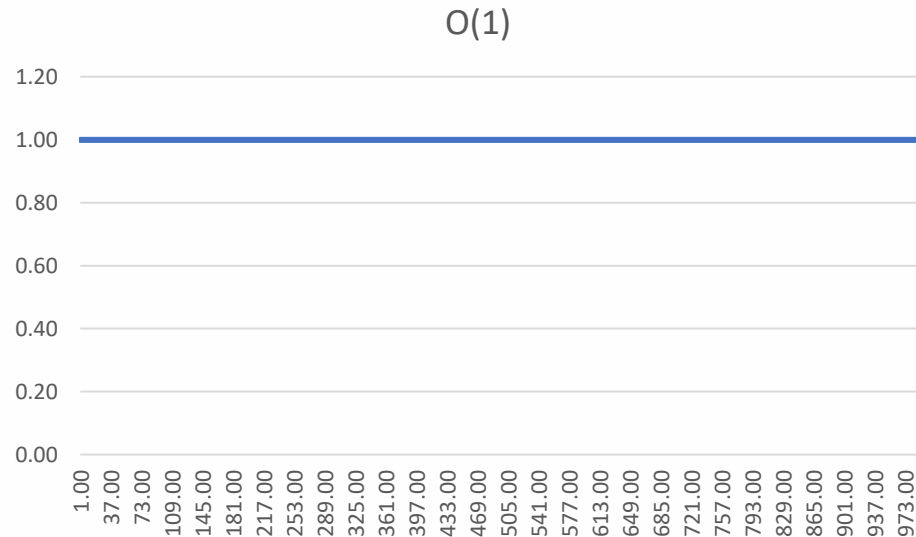
Growth Rates

- Algorithms analysis is all about understanding growth rates. That is **as the amount of data gets bigger, how much more resource will my algorithm require?**
- Typically, we describe the resource growth rate of a piece of code in terms of a function. To help understand the implications, this section will look at graphs for different growth rates from most efficient to least efficient.

GROWTH RATES

Constant Growth Rate

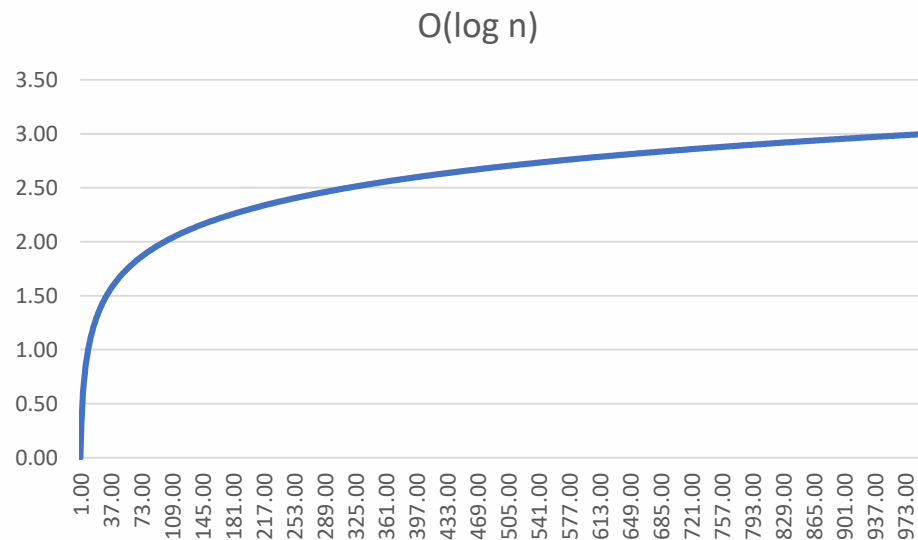
- A constant resource need is one where the resource need does not grow. That is processing 1 piece of data takes the same amount of resource as processing 1 million pieces of data. The graph of such a growth rate looks like a horizontal line.



File Manger: [BigO-GrowthRates.xlsx](#)

Logarithmic Growth Rate

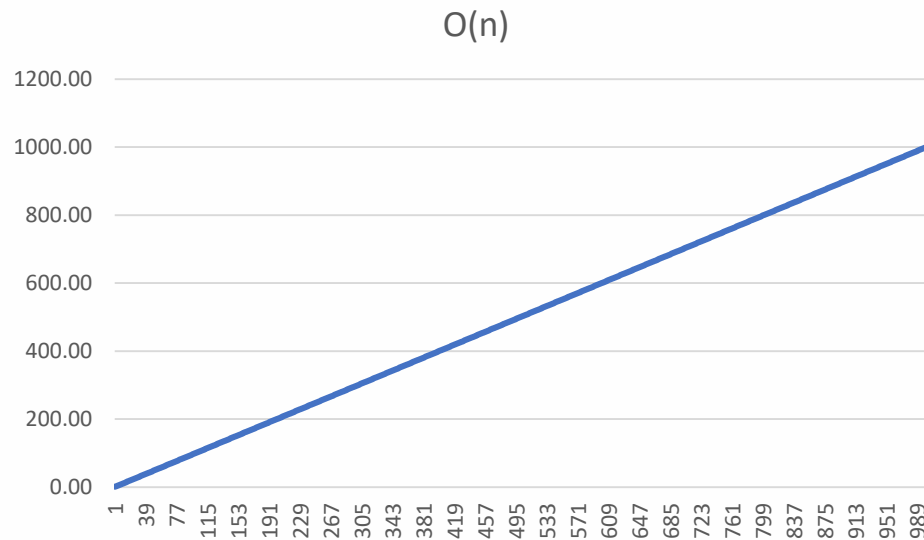
- A logarithmic growth rate is a growth rate where the resource needs grows by one unit each time the data is doubled. This effectively means that as the amount of data gets bigger, the curve describing the growth rate gets flatter (closer to horizontal but never reaching it). The following graph shows what a curve of this nature would look like.



GROWTH RATES

Linear Growth Rate

- A linear growth rate is a growth rate where the resource needs and the amount of data is directly proportional to each other. That is the growth rate can be described as a straight line that is not horizontal.

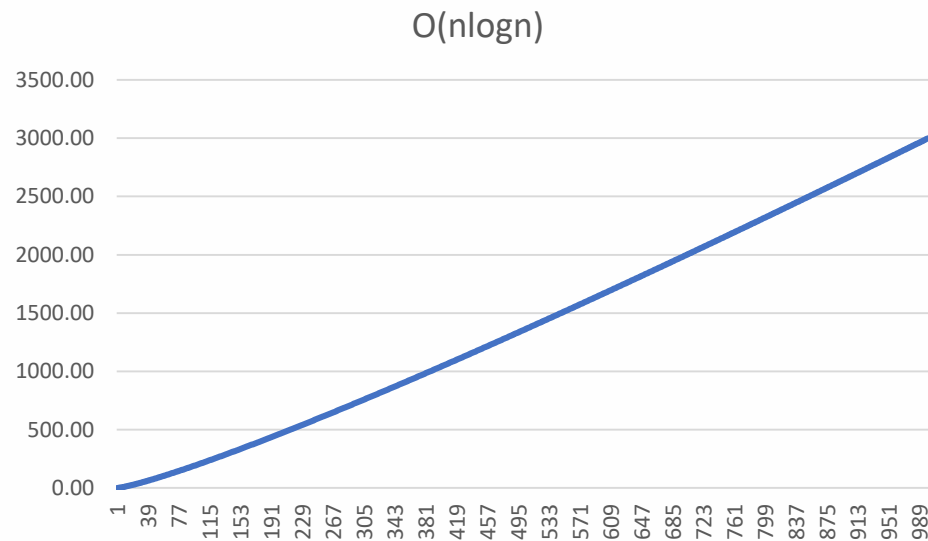


File Manger: [BigO-GrowthRates.xlsx](#)

GROWTH RATES

Log Linear

- A loglinear growth rate is a slightly curved line. the curve is more pronounced for lower values than higher ones.

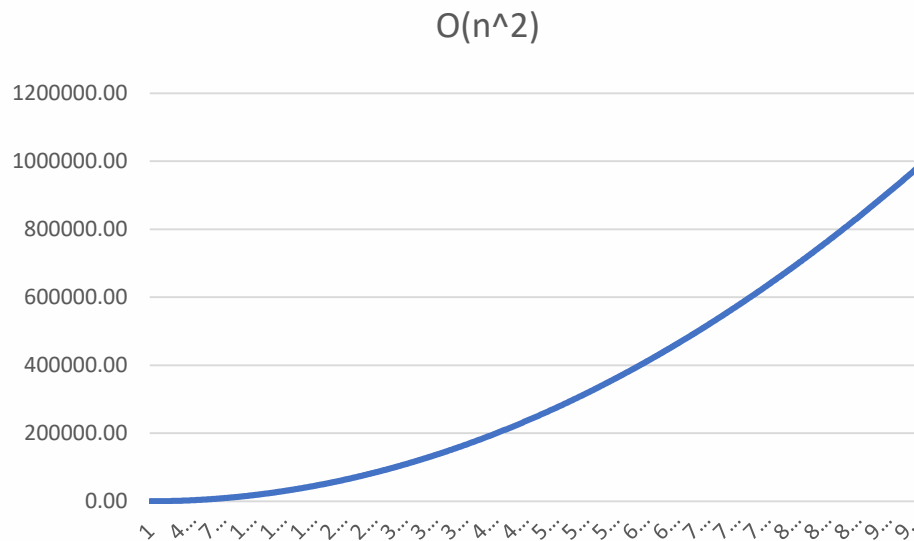


File Manger: [BigO-GrowthRates.xlsx](#)

GROWTH RATES

Quadratic Growth Rate

- A quadratic growth rate is one that can be described by a parabola.

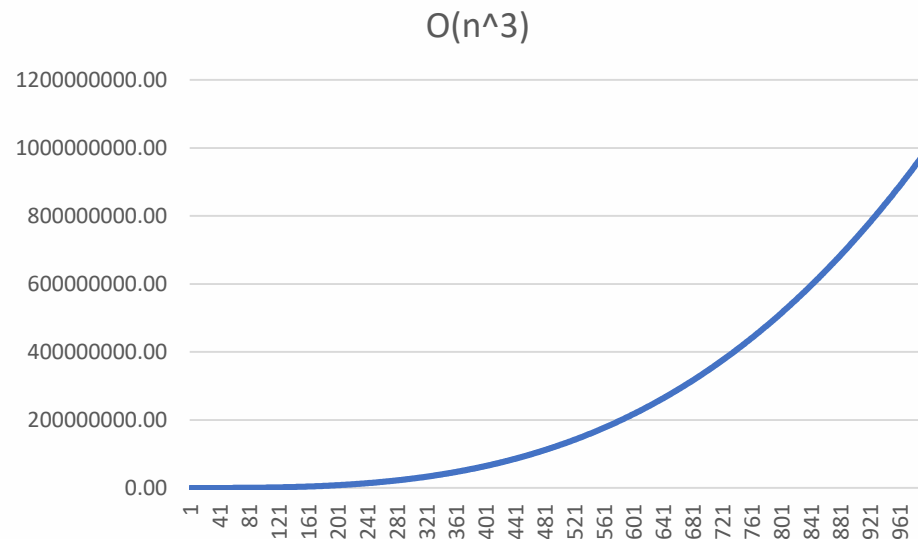


File Manger: [BigO-GrowthRates.xlsx](#)

GROWTH RATES

Cubic Growth Rate

- While this may look very similar to the quadratic curve, it grows significantly faster.

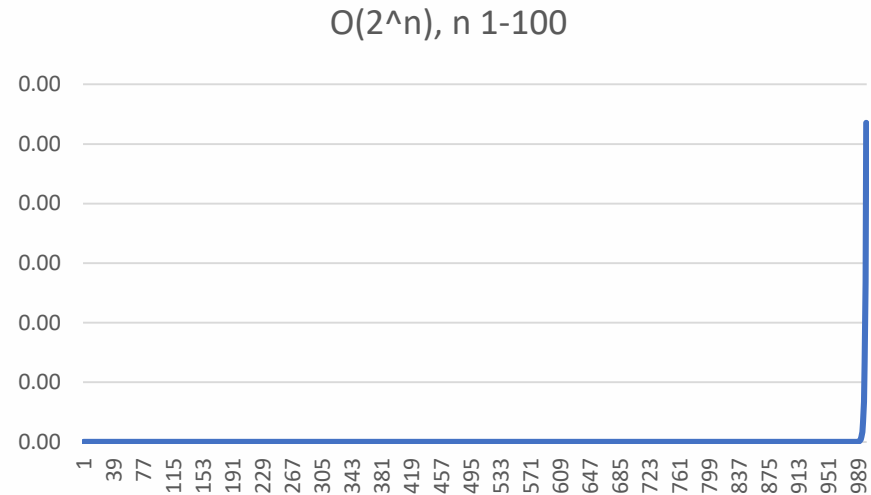
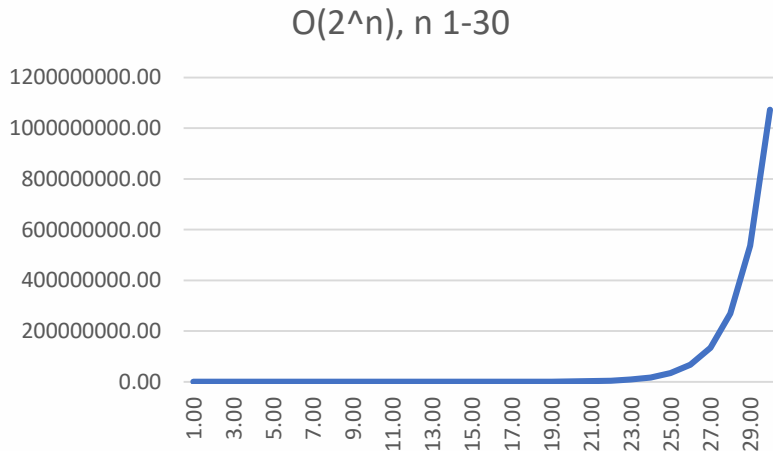


File Manger: [BigO-GrowthRates.xlsx](#)

GROWTH RATES

Exponential Growth Rate

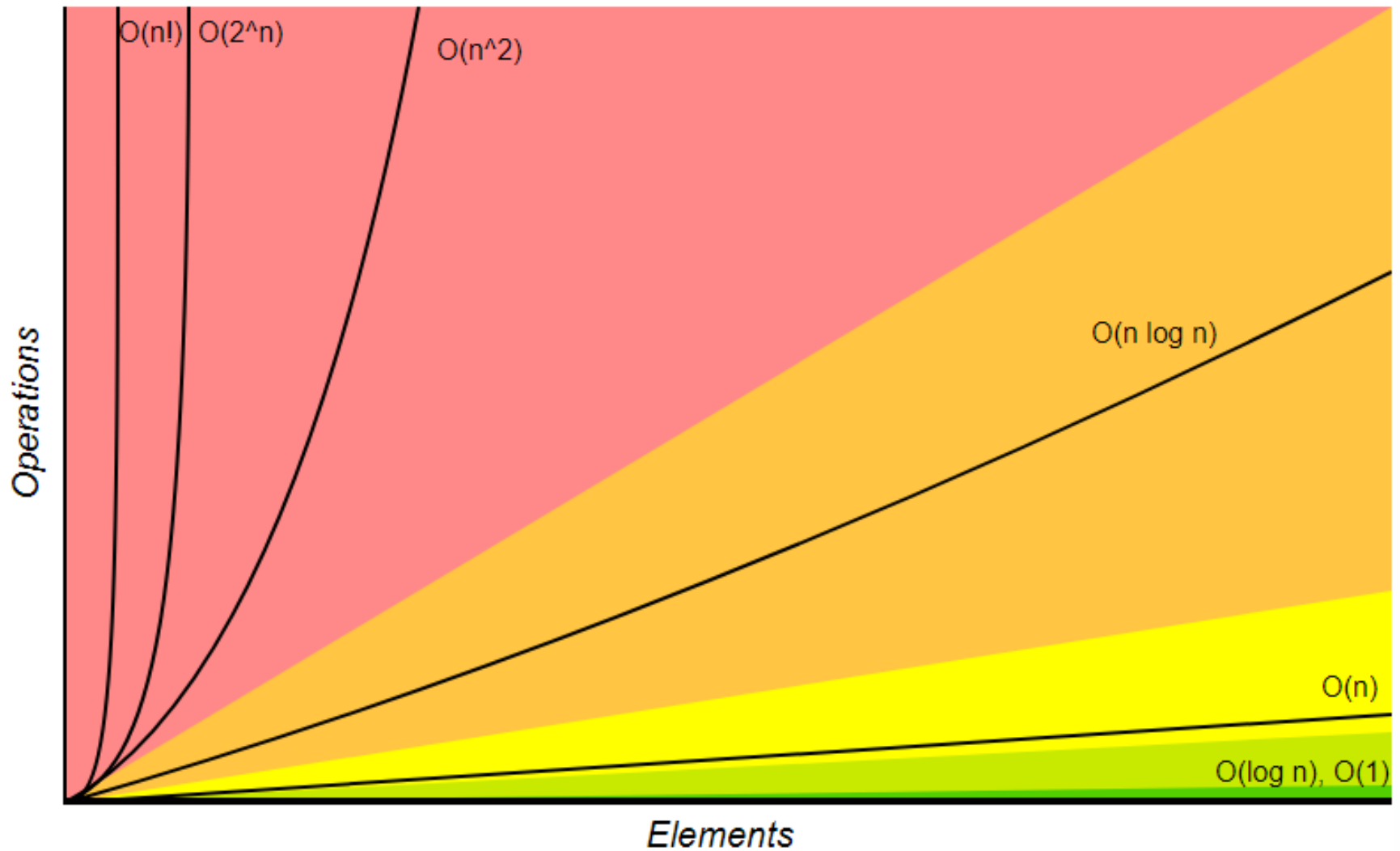
- An exponential growth rate is one where each extra unit of data requires a doubling of resource. As you can see the growth rate starts off looking like it is flat but quickly shoots up to near vertical (note that it can't actually be vertical)



File Manger: [BigO-GrowthRates.xlsx](#)

Big-O Complexity Chart

Horrible Bad Fair Good Excellent



<http://bigocheatsheet.com/>

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
<u>Array</u>	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<u>Stack</u>	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$
<u>Queue</u>	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$
<u>Singly-Linked List</u>	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$
<u>Doubly-Linked List</u>	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$
<u>Skip List</u>	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
<u>Hash Table</u>	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<u>Binary Search Tree</u>	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<u>Cartesian Tree</u>	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<u>B-Tree</u>	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
<u>Red-Black Tree</u>	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
<u>Splay Tree</u>	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
<u>AVL Tree</u>	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
<u>KD Tree</u>	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

<http://bigocheatsheet.com/>

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

<http://bigocheatsheet.com/>

COMPUTATIONAL COMPLEXITY



Computational Complexity

- Basically means *how **hard** is that problem?*



Hardest problems that we can possibly have?



Computational Complexity

- Basically means *how **hard** is that problem?*

Halting
problem



EASY  HARD

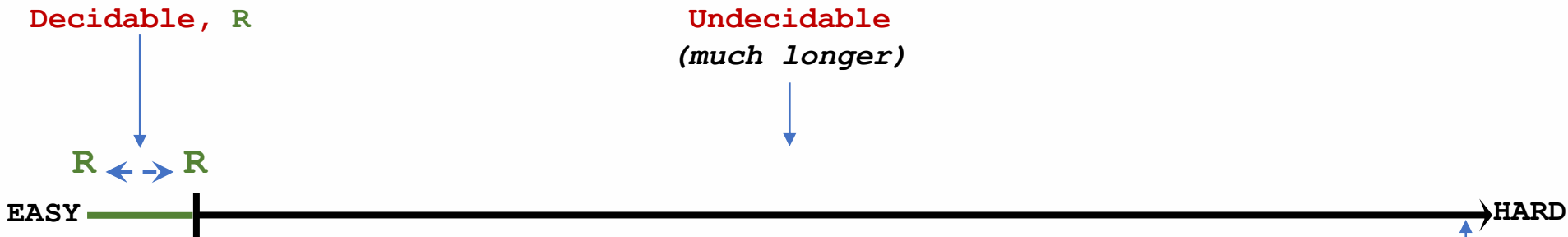
Halting problem: In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run forever.

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof was a mathematical definition of a computer and program, which became known as a Turing machine; the halting problem is undecidable over Turing machines. It is one of the first examples of a decision problem. --- https://en.wikipedia.org/wiki/Halting_problem

Terminate or Not? Given a program, does it ever halt/stop running? For all programs **in finite time**.



Computational Complexity, *how hard?*

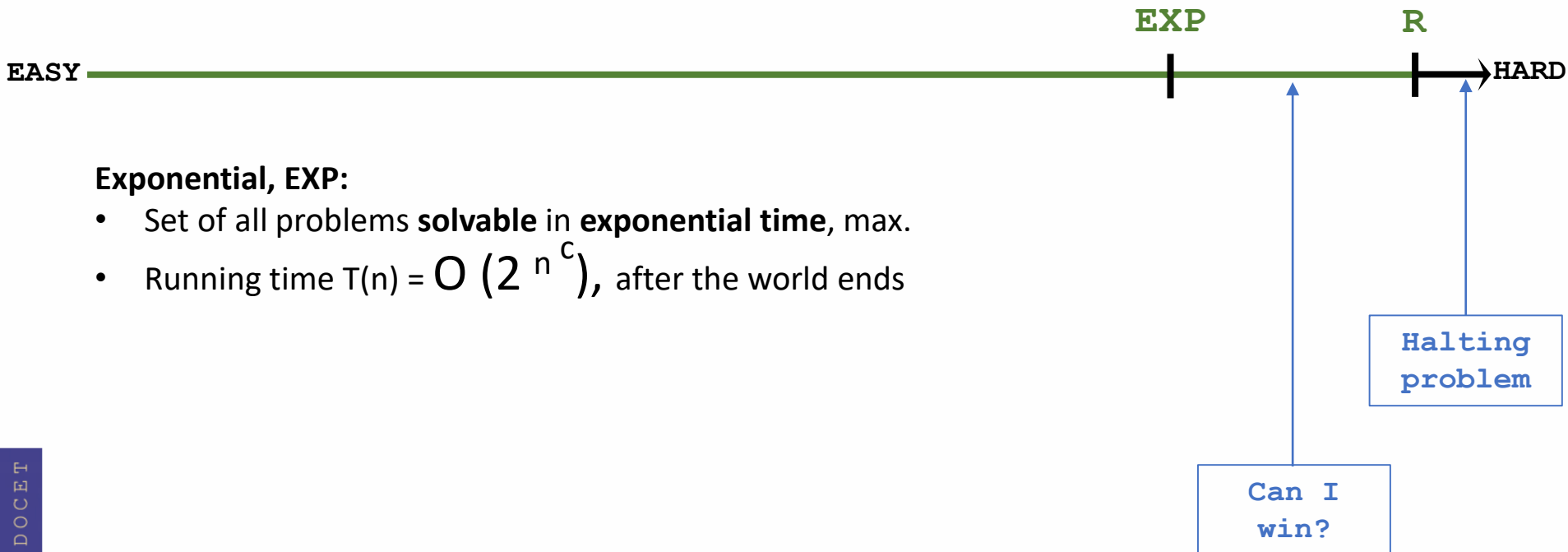


Decidable or R:

- Set of all problems solvable in **finite time**.
- Or there exists an algorithm to compute the answer in a **finite time**.
- R = “Recursive” or it will terminate at some point in time.
- Not just Halting problem, but almost all the **decision** problems are not in R.



Computational Complexity, *how hard?*





Computational Complexity, *how hard?*

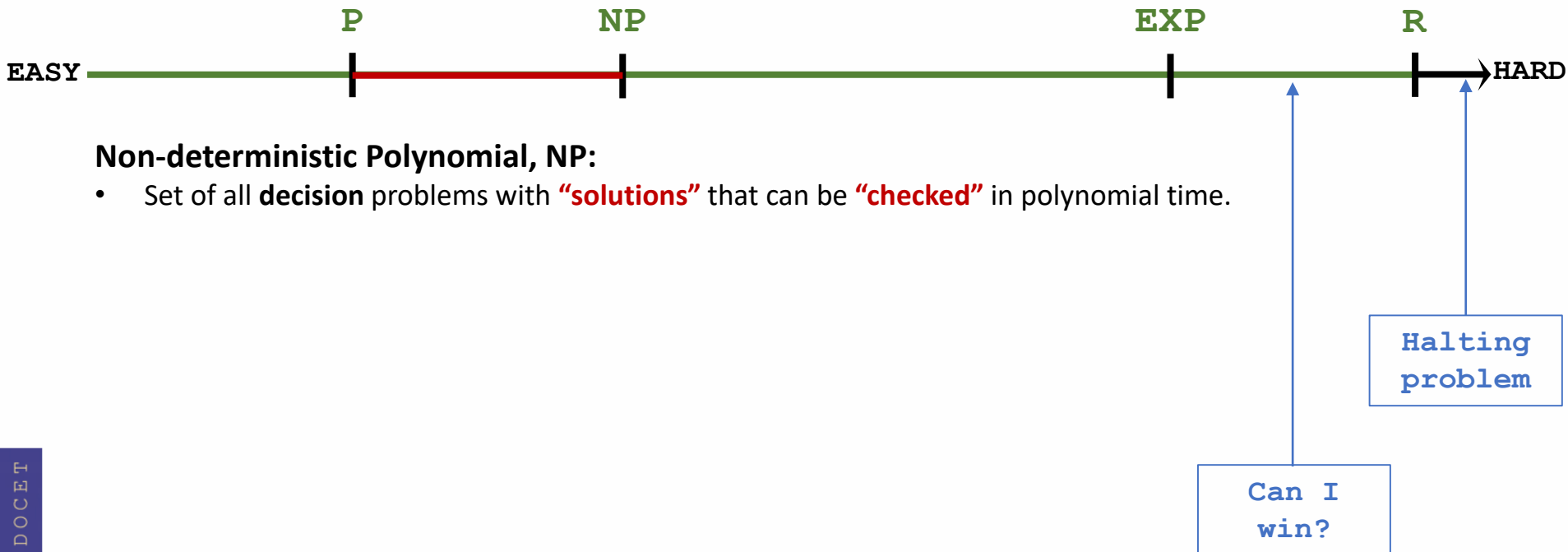


Non-deterministic Polynomial, NP:

- Set of all problems **solvable** in **polynomial time**, max, via a **“lucky” algorithm**.
- Running time $T(n) = O(n^c)$, polynomial in terms of input size
- NP means it is possible to write a verifier that takes a solution (and the input) to the problem and output Correct or Not Correct (with reasoning, a string) .
 - The verifier runs in P, polynomial time.
 - Problem input (the solution) must be P in size (otherwise the verifier cannot consume it fast enough).
 - Problem output is also in P size as well.
- The chance of getting a proof like this for a NP problem is almost zero. We assume that we are always lucky and always get a proof. 1 guess = 1 right choice. It is a “magical” machine.
- This magical machine is the only possible proof that $P \neq NP$. At least \$1 million to prove **$N == NP$** or **$N \neq NP$** .



Computational Complexity, *how hard?*



Non-deterministic Polynomial, NP:

- Set of all **decision** problems with "**solutions**" that can be "**checked**" in polynomial time.



Computational Complexity, *how hard?*



Polynomial, P:

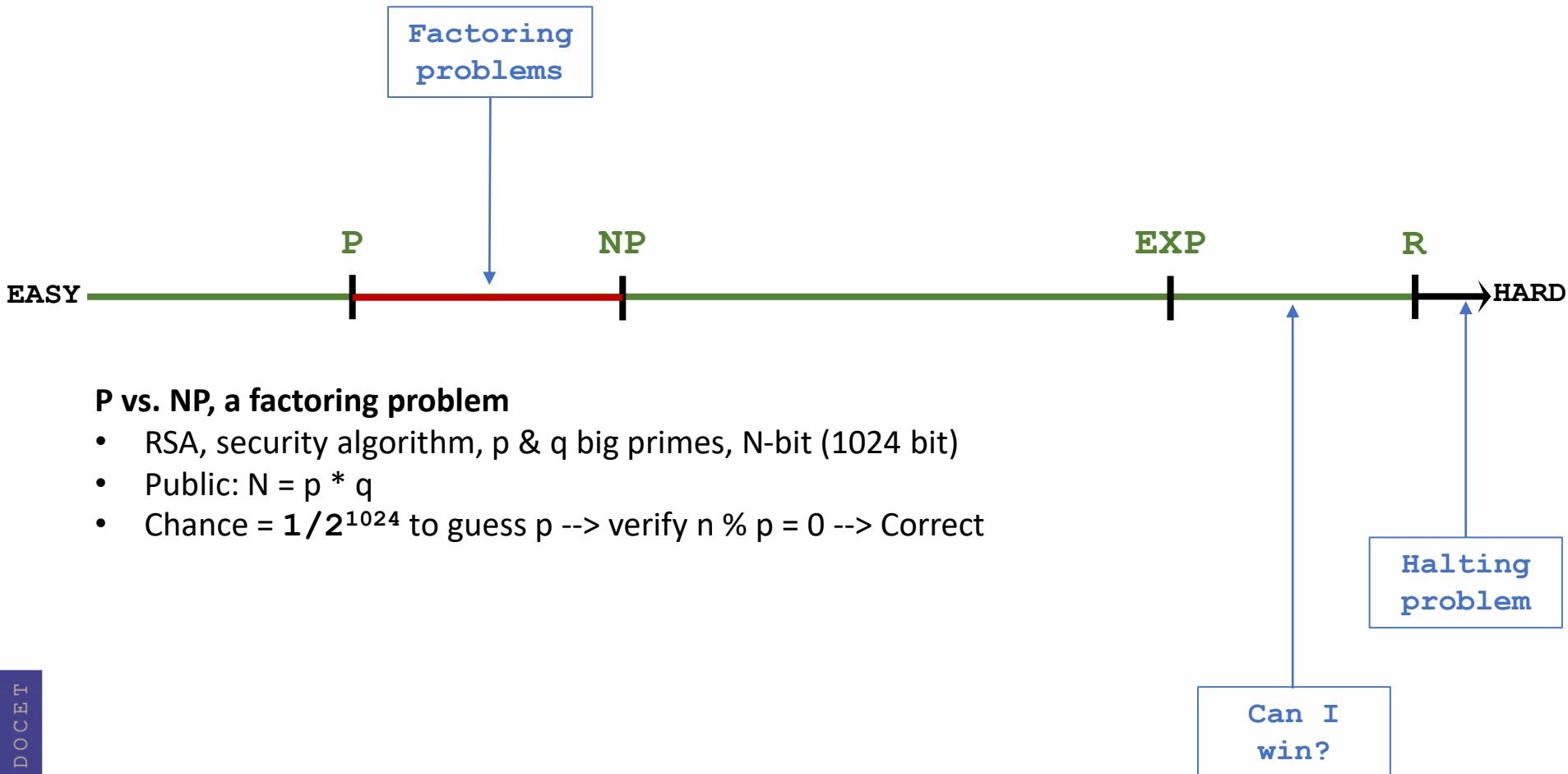
- Set of all problems **solvable** in **polynomial time**, max.
- Running time $T(n) = O(n^c)$, polynomial in terms of input size
- Double the input $2 * n$, then running time $T(n) = O((2n)^c) = O(2^c * n^c)$
- Input doubled, running time increased by constant a constant factor 2^c
- Exponential with input doubled, then quadratically increased (not linear)
- Best (vs. NP, EXP...)

Can I
win?

Halting
problem



Computational Complexity, how *hard*?

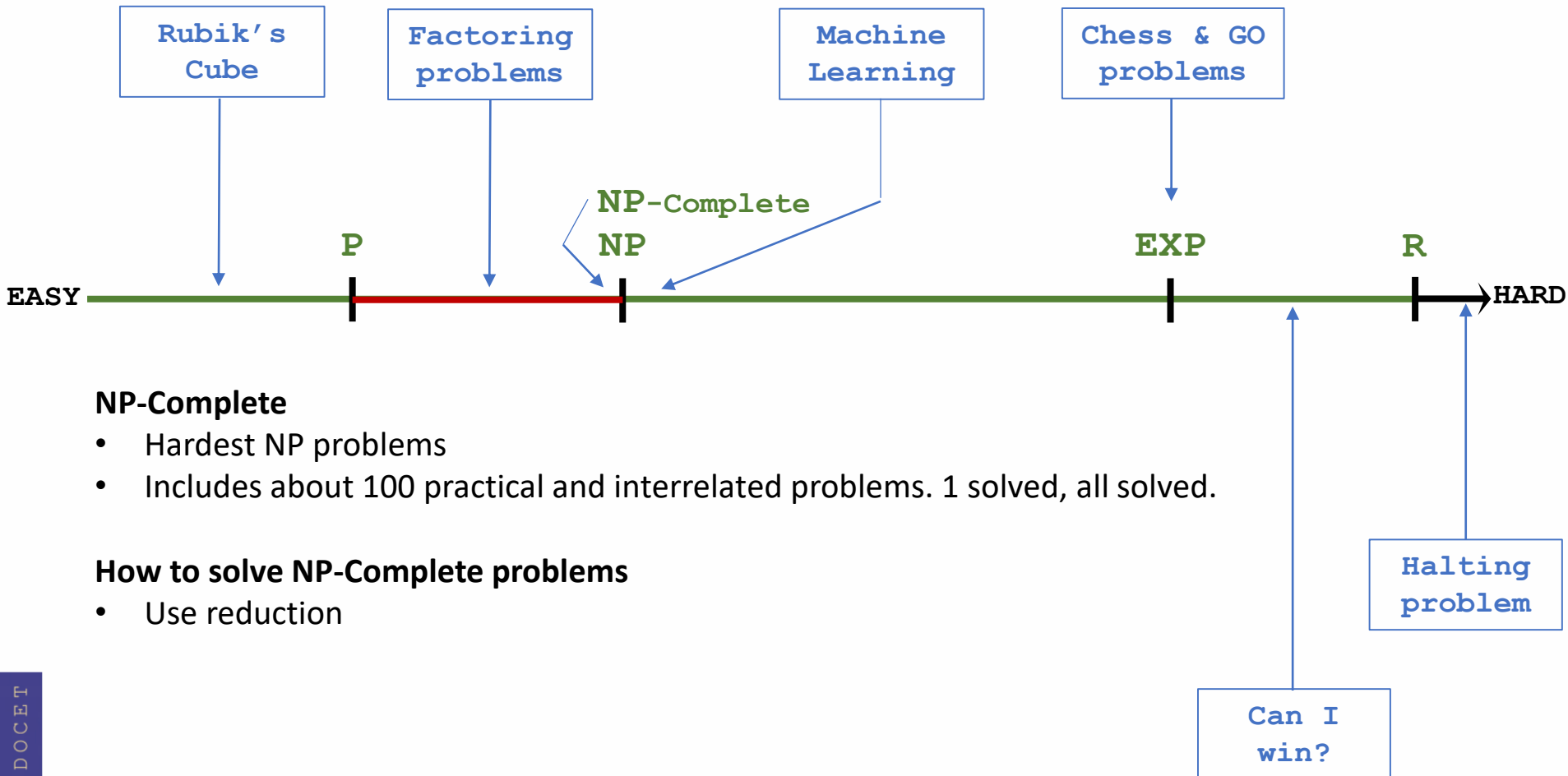


P vs. NP, a factoring problem

- RSA, security algorithm, p & q big primes, N -bit (1024 bit)
- Public: $N = p * q$
- Chance = $1/2^{1024}$ to guess $p \rightarrow$ verify $n \% p = 0 \rightarrow$ Correct



Computational Complexity, how *hard*?



NP-Complete

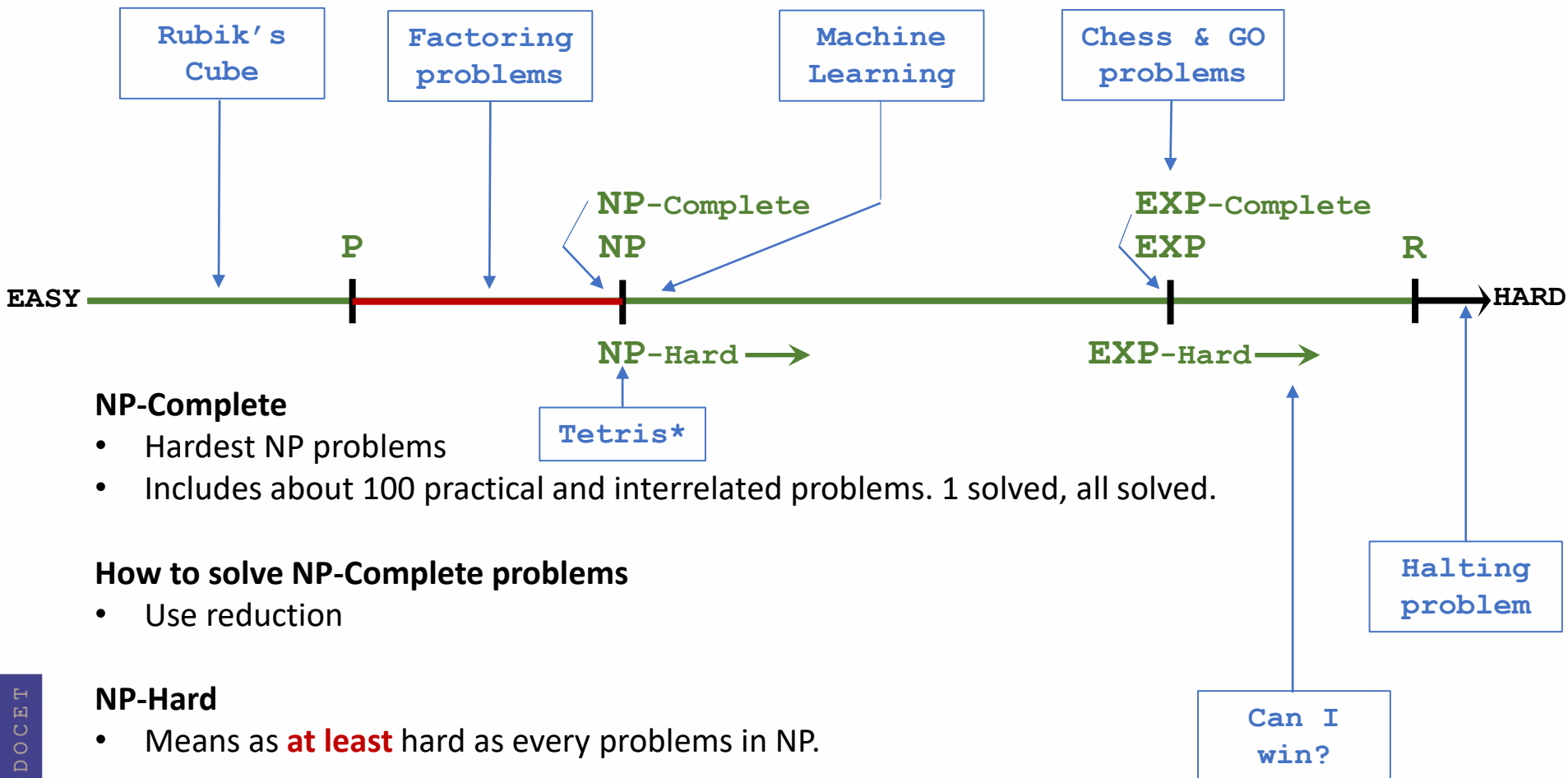
- Hardest NP problems
- Includes about 100 practical and interrelated problems. 1 solved, all solved.

How to solve NP-Complete problems

- Use reduction



Computational Complexity, how *hard*?



NP-Complete

- Hardest NP problems
- Includes about 100 practical and interrelated problems. 1 solved, all solved.

How to solve NP-Complete problems

- Use reduction

NP-Hard

- Means as **at least** hard as every problems in NP.

Unknown?

- $P == NP$? and even $NP\text{-hard} == EXP\text{-hard}$?

<< *P* vs. *NP* VIDEO CLIP >>

See you next class!