COMP 251

Computational Complexity

Goals for This Lecture

- Describe the various behaviors of simple analytic functions.

Goals for This Lecture

- Describe the various behaviors of simple analytic functions.
- Define time and space complexity.

Goals for This Lecture

- Describe the various behaviors of simple analytic functions.
- Define time and space complexity.
- Show how certain problems (algorithms) behave intractably.

Goals for This Lecture

- Describe the various behaviors of simple analytic functions.
- Define time and space complexity.
- Show how certain problems (algorithms) behave intractably.
- Define the various classes of complexity.,

Goals for This Lecture

- Describe the various behaviors of simple analytic functions.
- Define time and space complexity.
- Show how certain problems (algorithms) behave intractably.
- Define the various classes of complexity.,
- Define the fundamental Data Structures and their Computational Complexities.

Goals for This Lecture

- Describe the various behaviors of simple analytic functions.
- Define time and space complexity.
- Show how certain problems (algorithms) behave intractably.
- Define the various classes of complexity.,
- Define the fundamental Data Structures and their Computational Complexities.
- Briefly discuss some philosophical implications.

Definition

Computational complexity theory:

The branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their **inherent difficulty**, and relating those classes to each other.

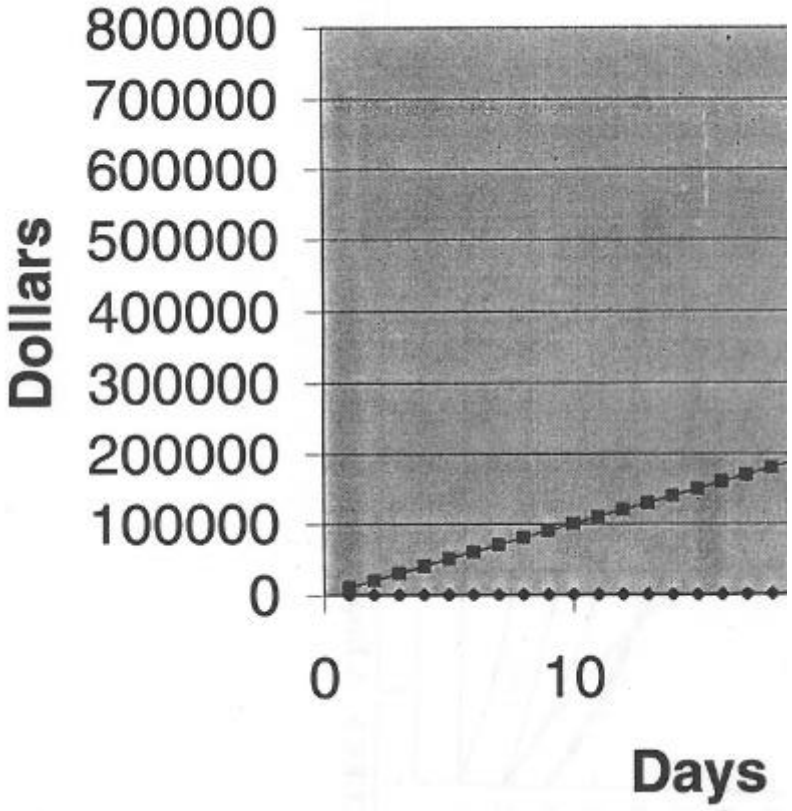
Question

You have just won the lottery.

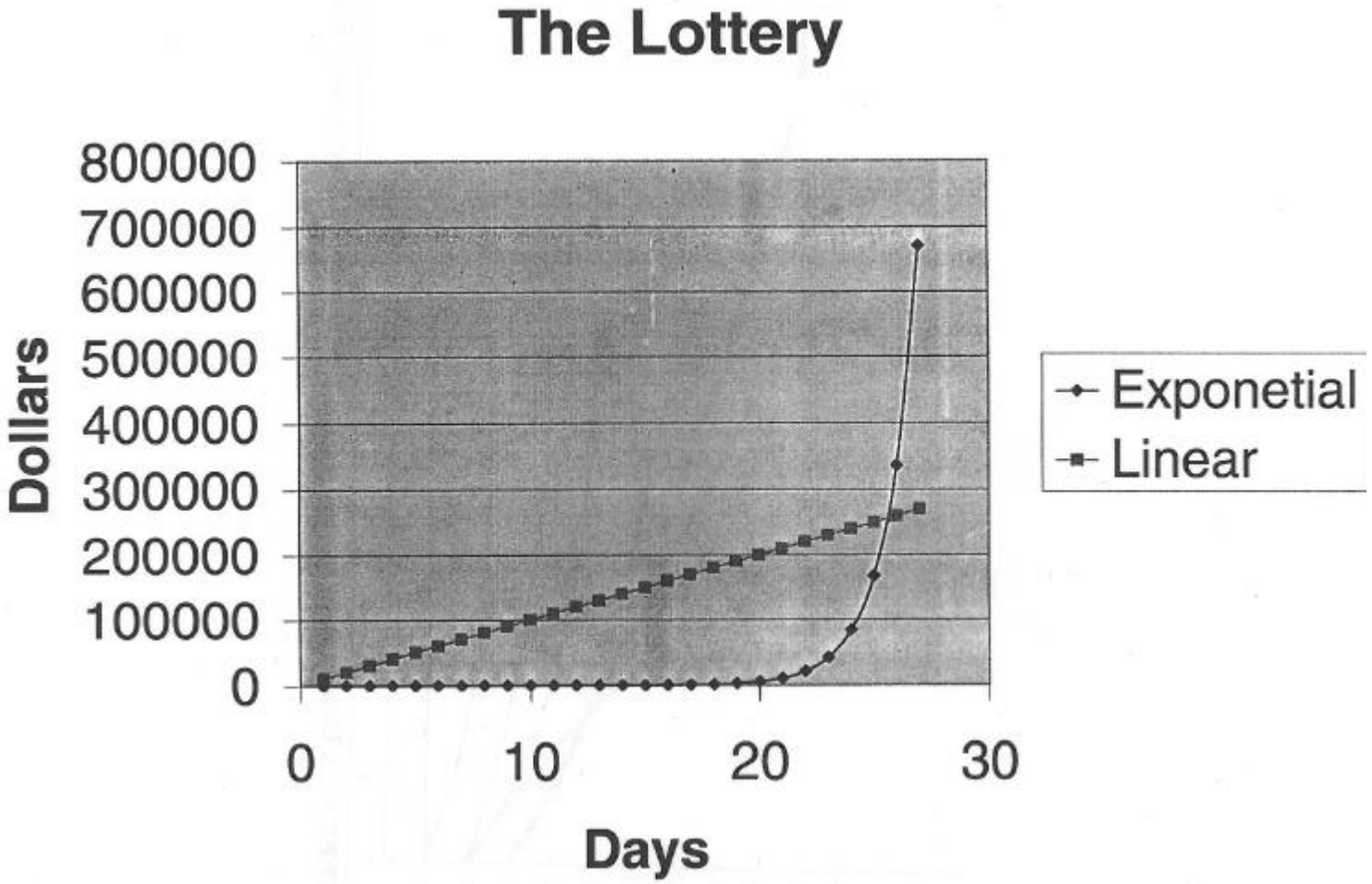
You are given the choice of one of the following payment options:

1. **$1000/day for the rest of your life, or**
2. **1 cent today, 2 cents tomorrow, 4 cents the day after, and so on.**

Question

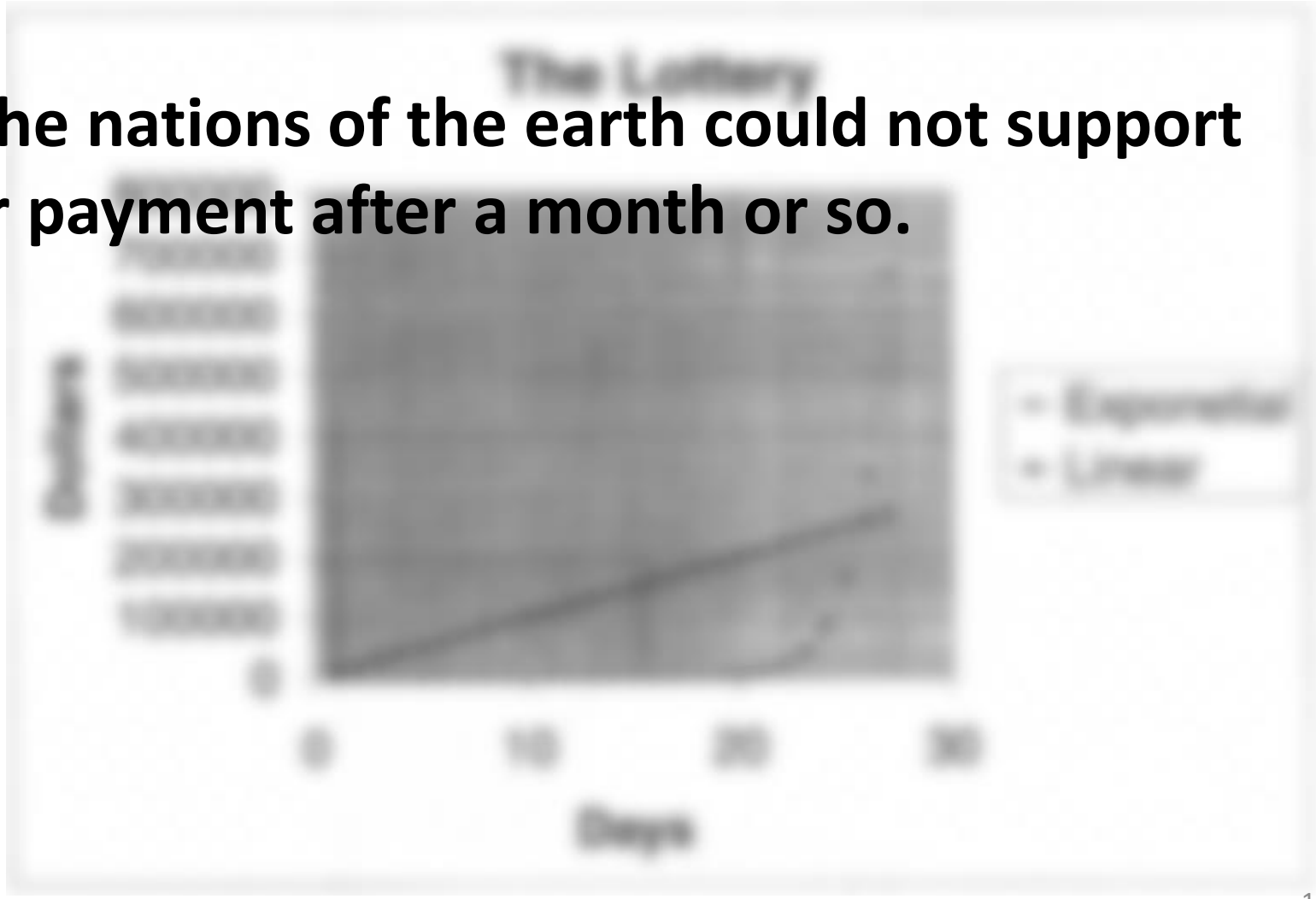

800000
700000
600000
500000
400000
300000
200000
100000
0
Dollars
0
10
Days

Question


The Lottery
800000
700000
600000
500000
400000
300000
200000
100000
0
Dollars
0
10
20
30
Days
Exponetial
Linear

Question

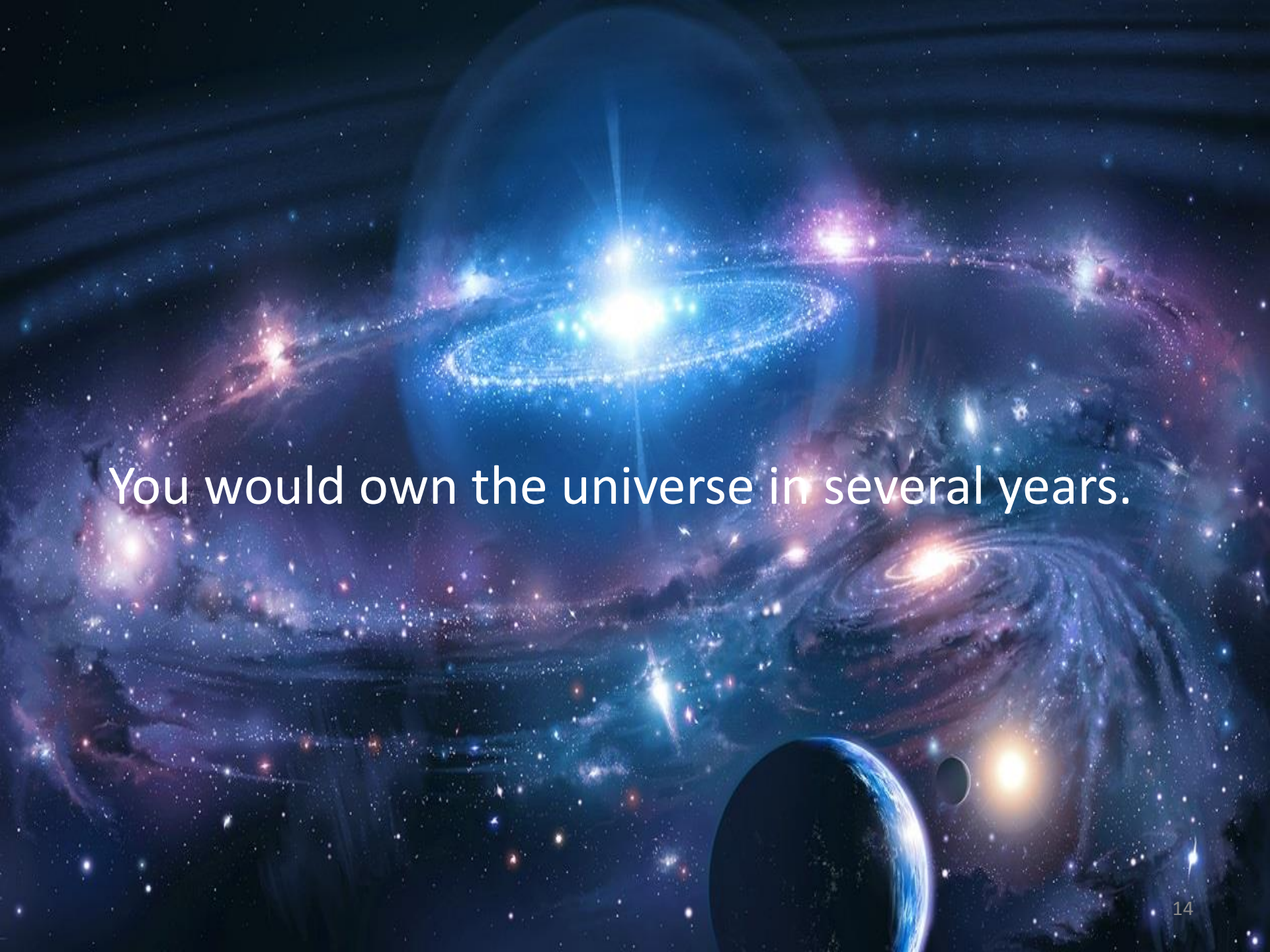
All the nations of the earth could not support your payment after a month or so.



Question

All the nations of the earth could not support your payment after a month or so.

You would own everything in under two months.


You would own the universe in several years.

Recall the Different Types of "Well-Known" Functions

Given a constant, n, and a variable, x, we can construct the following:

- x^n – **Mono**mial

Recall the Different Types of "Well-Known" Functions

Given a constant, n, and a variable, x, we can construct the following:

- x^n – **Mono**mial
- $P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a^1 x + a_0$ – **Poly**nomial
  - $P(x) = a_2 x^2 + a_1 x + a_0$ – Quadratic
  - $P(x) = a_1 x + a_0$ – Linear
  - $P(x) = a_0$ – Constant

Recall the Different Types of "Well-Known" Functions

Given a constant, *n*, and a variable, *x*, we can construct the following:

- x^n – **Mono**mial
- $P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a^1 x + a_0$ – **Poly**nomial
  - $P(x) = a_2 x^2 + a_1 x + a_0$ – Quadratic
  - $P(x) = a_1 x + a_0$ – Linear
  - $P(x) = a_0$ – Constant
- n^x – Exponential
- $n!$ – Factorial
- And their inverse functions (such as $x^{1/n}$ or $\log x$)

NOTE: Inverse Factorial is Complicated

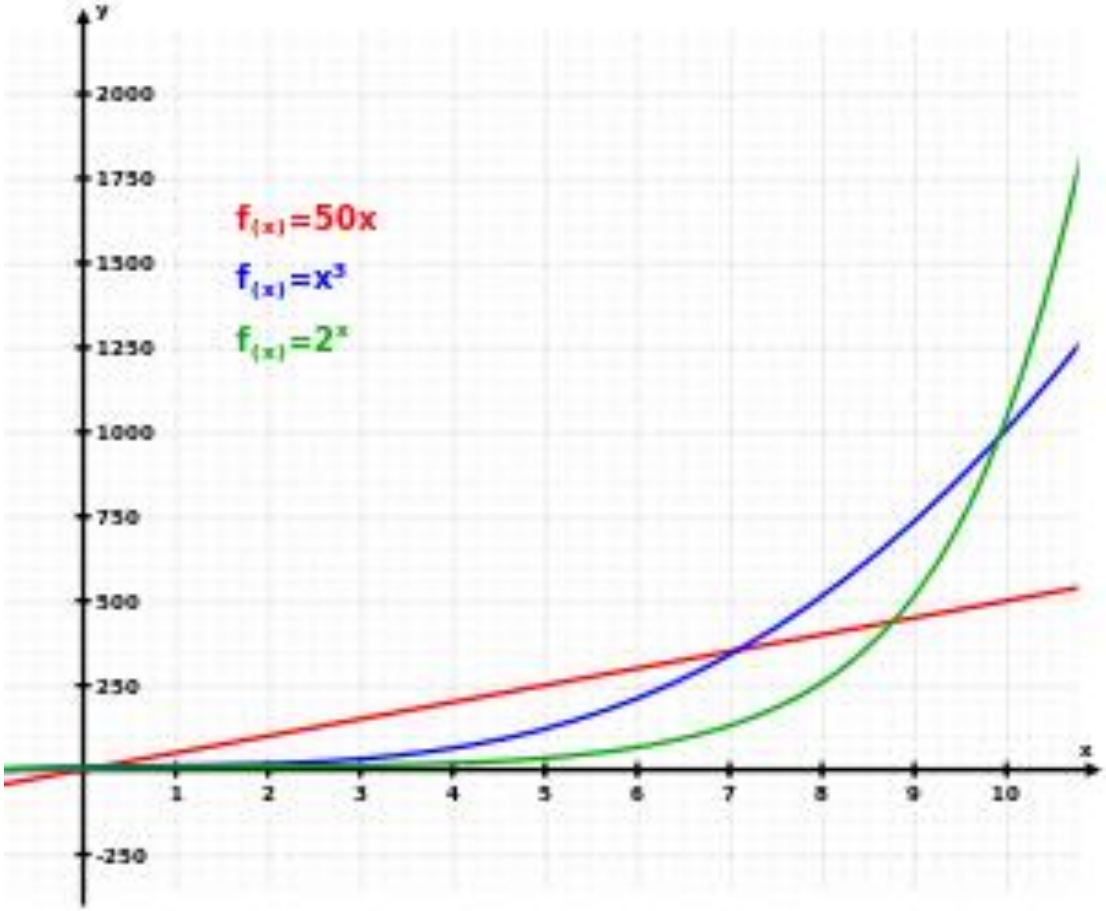
David Cantrell gives a good approximation of $\Gamma^{-1}(n)$ on this page.

I'll copy the result here in case that page ever goes down:

k = the positive zero of the digamma function, approximately 1.461632
$c = \sqrt{2\pi}/e - \Gamma(k)$, approximately 0.036534
$L(x) = \ln(\frac{x+c}{\sqrt{2\pi}})$
$W(x)$ = Lambert W function
$ApproxInvGamma(x) = L(x)/W(\frac{L(x)}{e}) + \frac{1}{2}$

http://mathoverflow.net/questions/12828/inverse-gamma-function

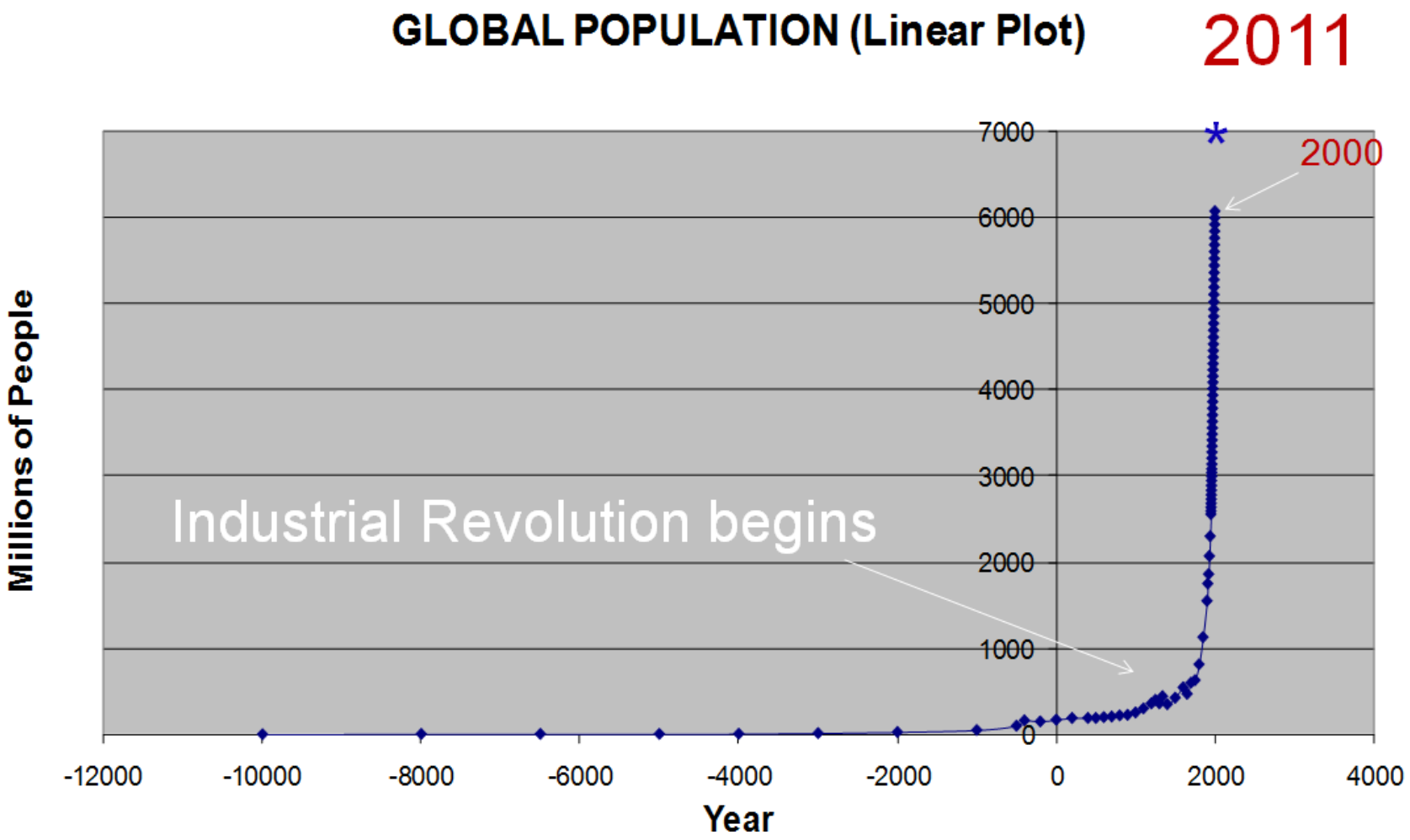
Different Kinds of Functional Behavior



Exponential functions grow extremely fast in comparison to polynomials

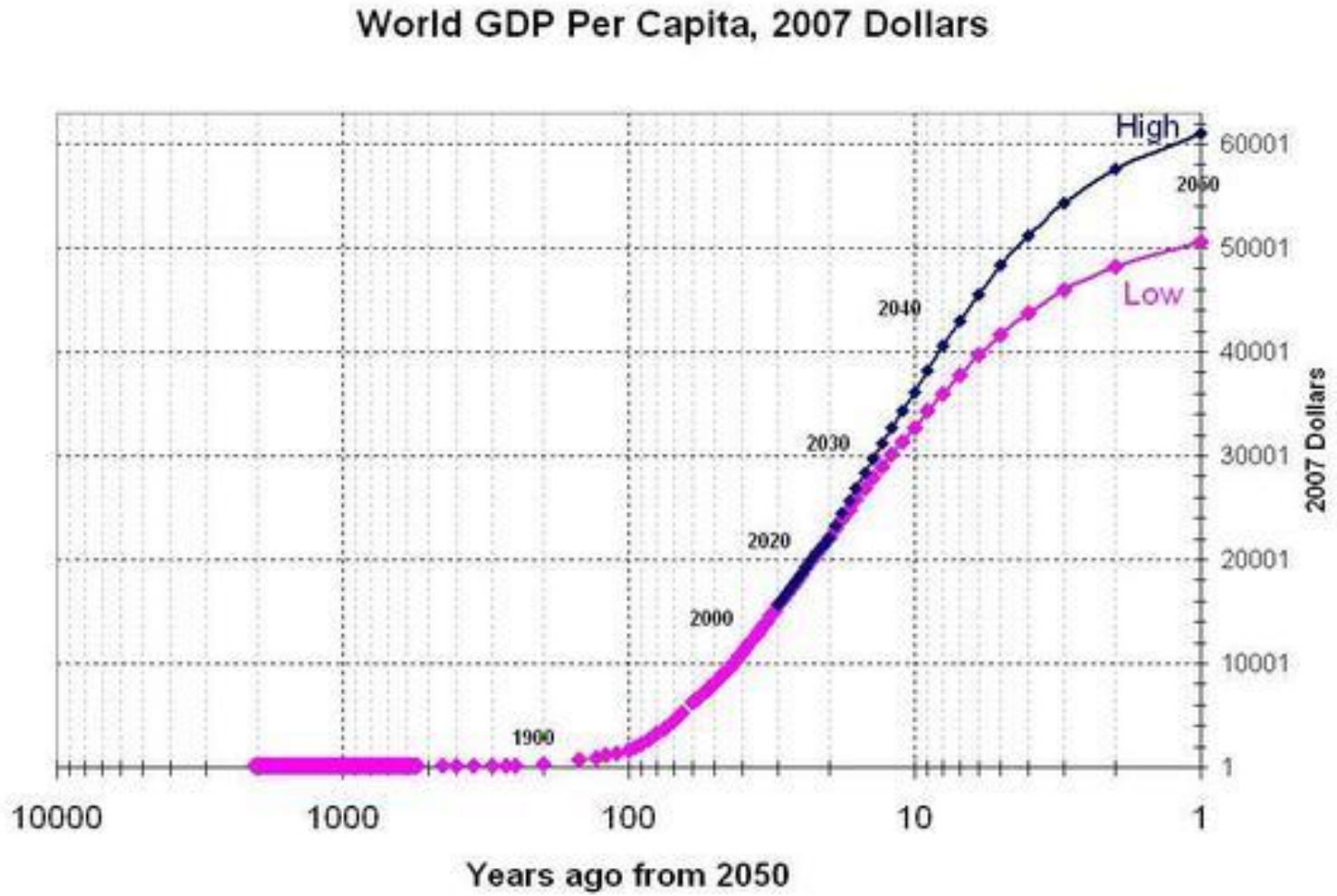
Exponential Growth

Exponential Growth



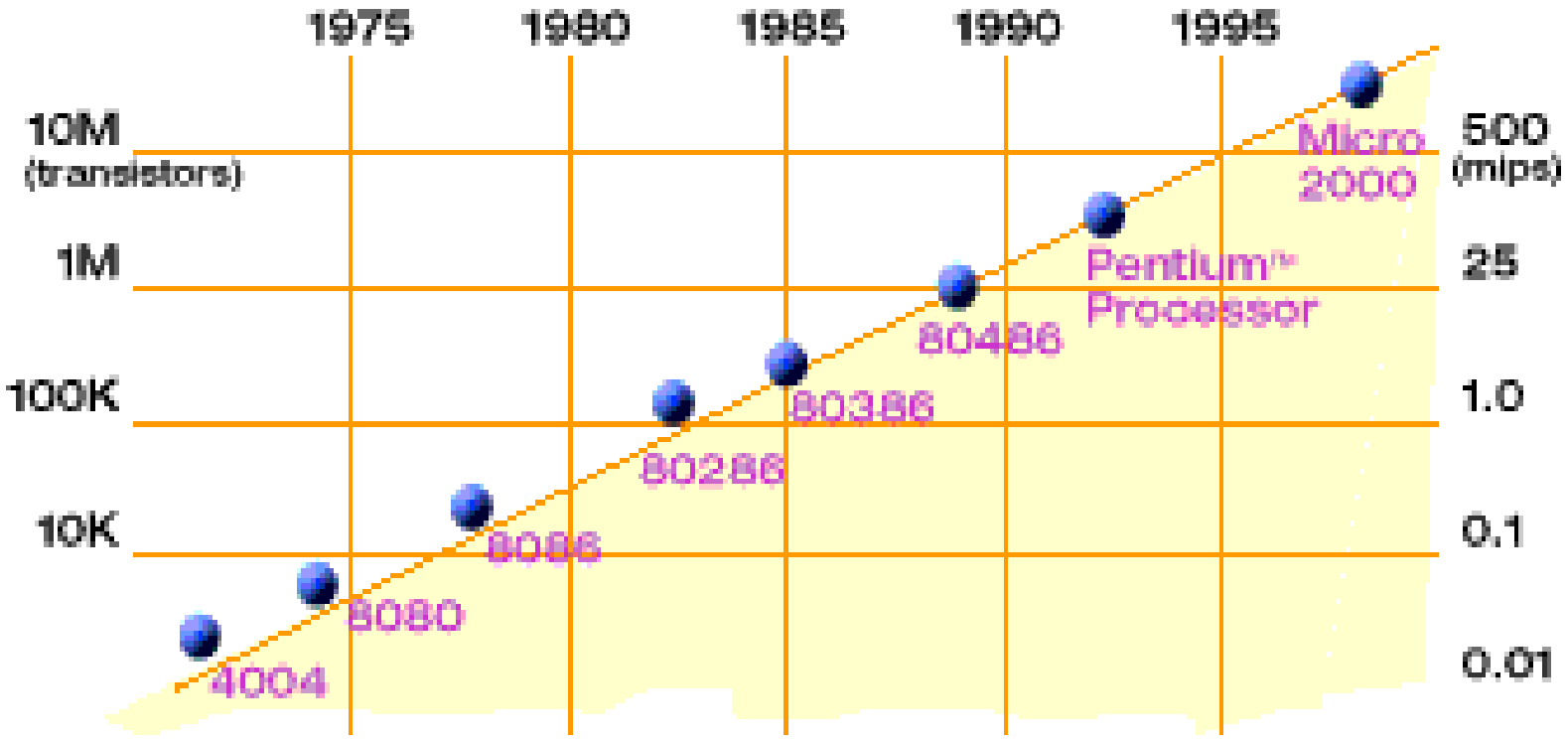
Currenrt Doubling Time: ~60 years

Exponential Growth



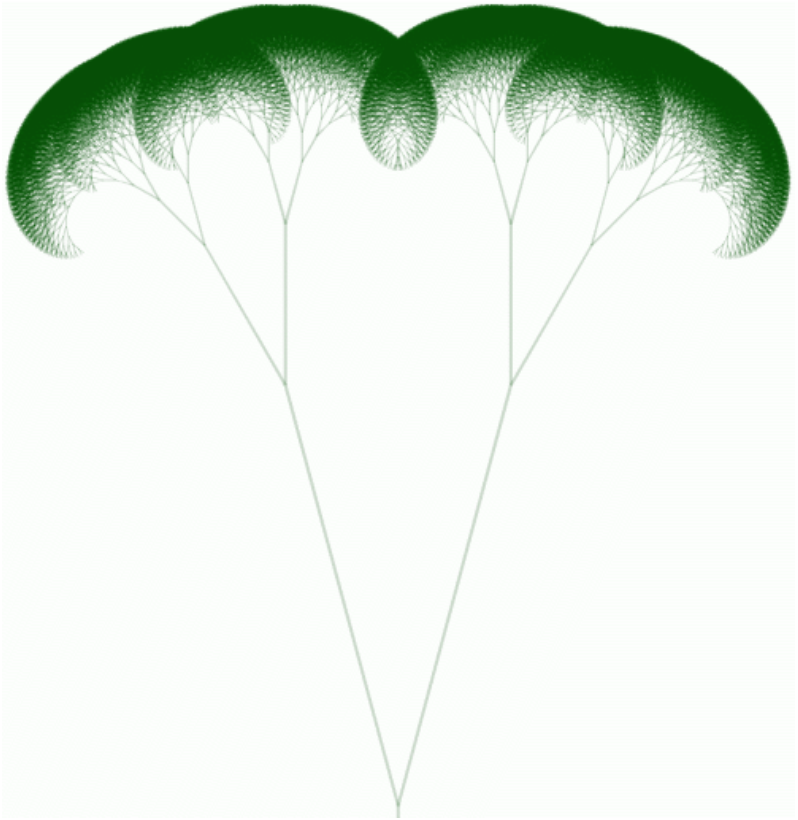
Exponential Growth

Moore's Law: Doubling Time: 24 months



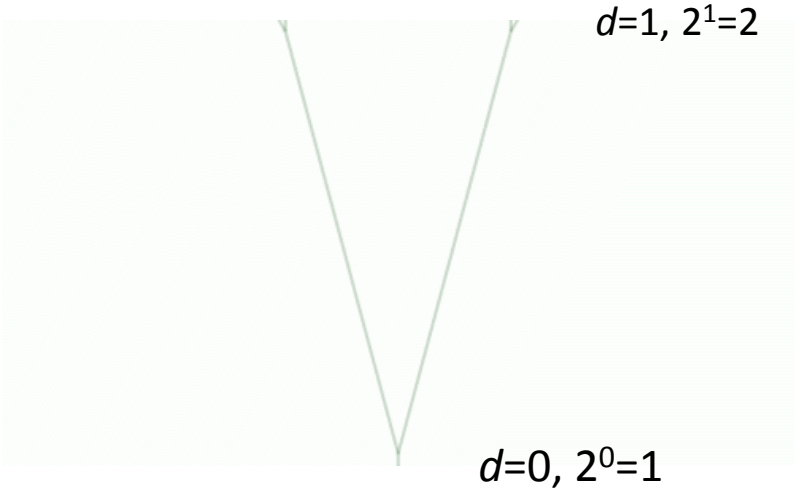
Binary Trees

- Every node branches into two descendant nodes (branching factor equals 2).



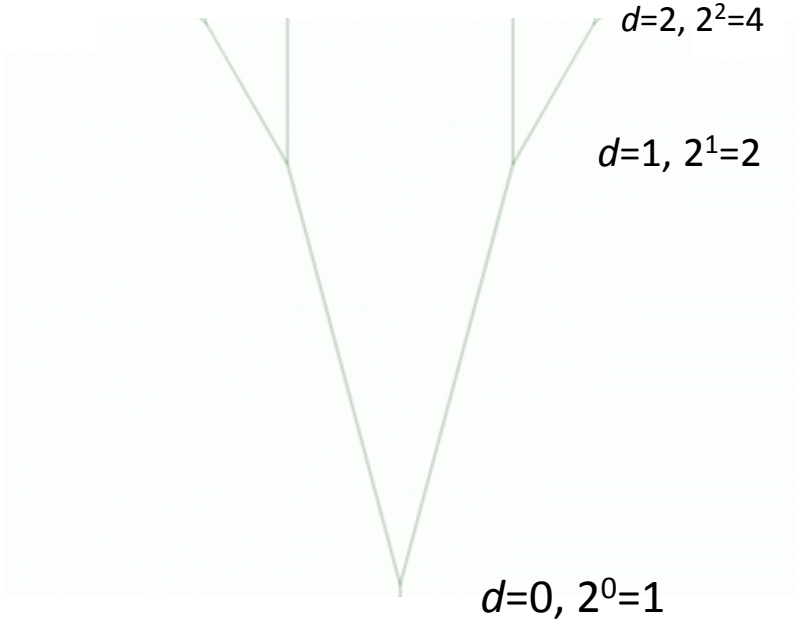
Binary Trees

- Every node branches into two descendant nodes (branching factor equals 2).
- The number of nodes expands exponentially with depth.



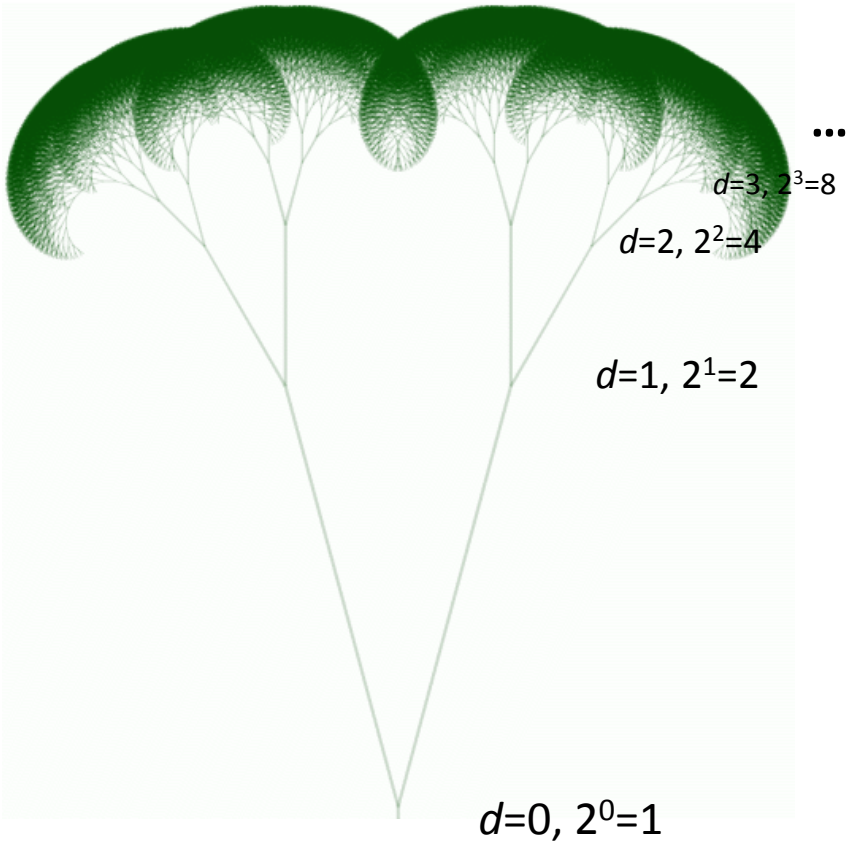
Binary Trees

- Every node branches into two descendant nodes (branching factor equals 2).
- The number of nodes expands exponentially with depth.

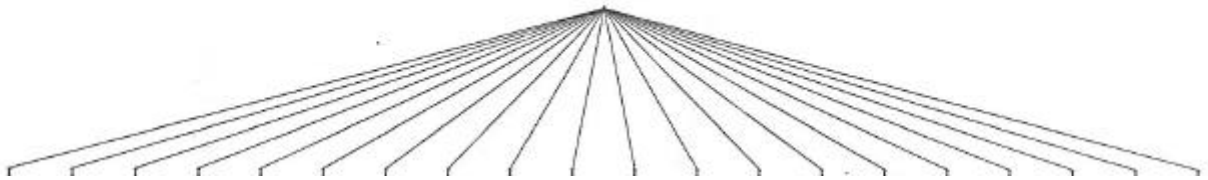


Binary Trees

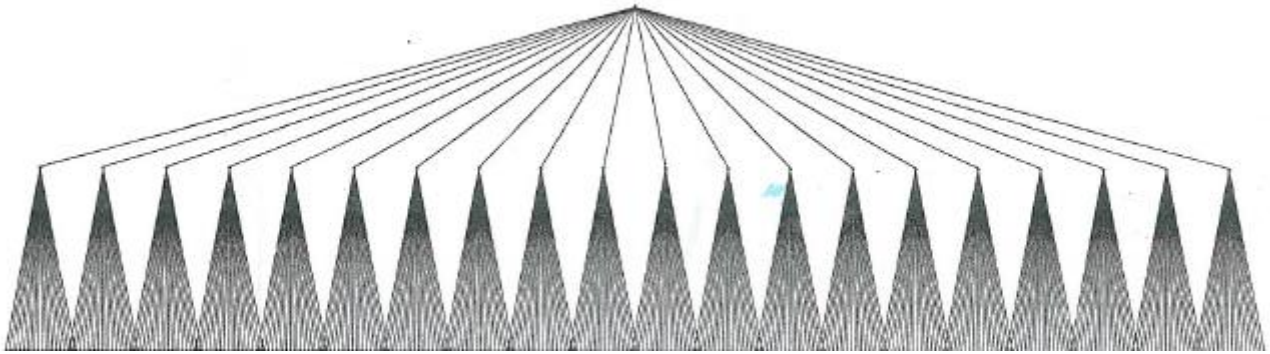
- Every node branches into two descendant nodes (branching factor equals 2).
- The number of nodes expands exponentially with depth.



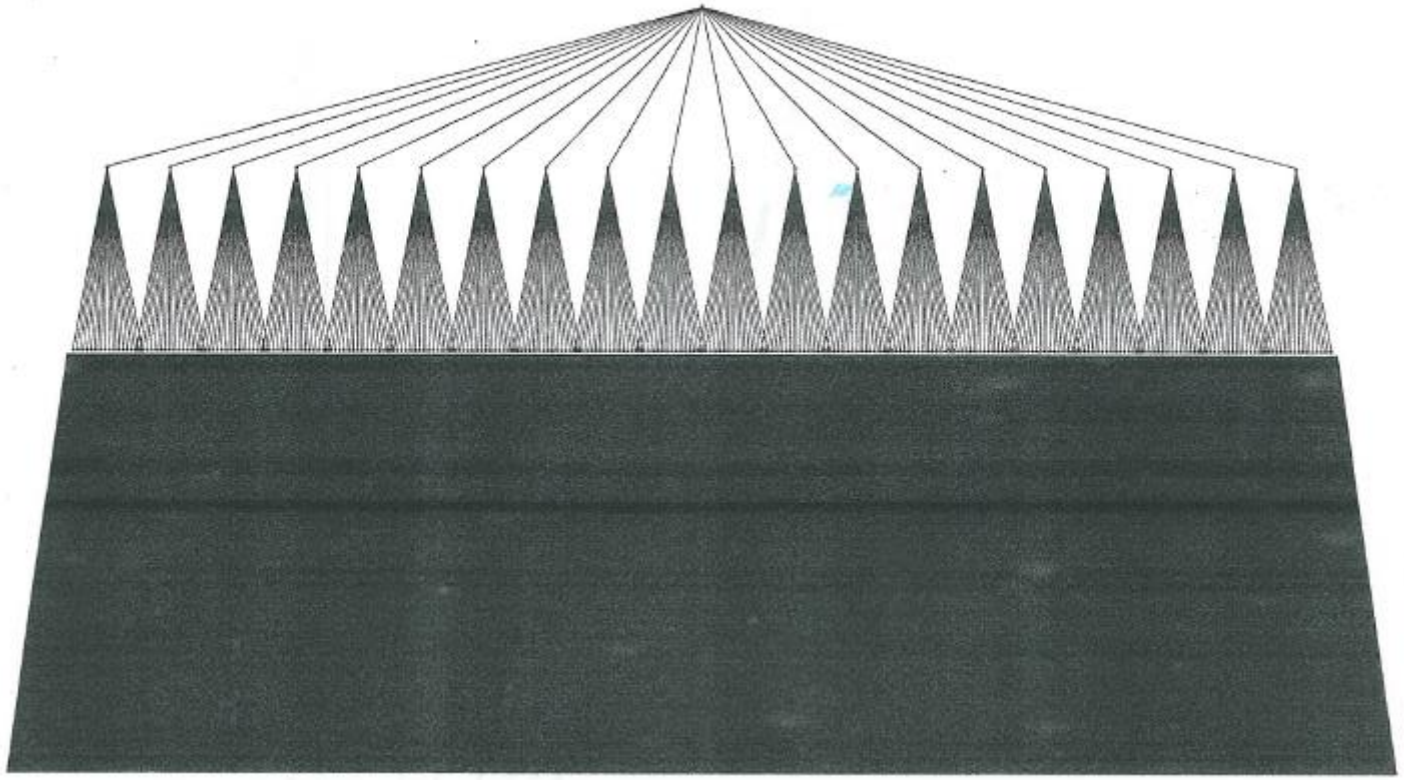
The Chess Game Tree



The Chess Game Tree



The Chess Game Tree



~35^d

Chess Versus Go

Chess		
Move	White	Black
1	20	700
2	24,500	857,500
3	30,012,500	1,050,437,500
4	36,765,312,500	1.29E+12
5	4.50E+13	1.58E+15
6	5.52E+16	1.93E+18
7	6.76E+19	2.37E+21
8	8.28E+22	2.90E+24
9	1.01E+26	3.55E+27
10	1.24E+29	4.35E+30
11	1.52E+32	5.33E+33
12	1.86E+35	6.53E+36
13	2.28E+38	7.99E+39
14	2.80E+41	9.79E+42
15	3.43E+44	1.20E+46
16	4.20E+47	1.47E+49
17	5.14E+50	1.80E+52
18	6.30E+53	2.21E+55
19	7.72E+56	2.70E+58
20	9.45E+59	3.31E+61
21	1.16E+63	4.05E+64
22	1.42E+66	4.97E+67
23	1.74E+69	6.08E+70
24	2.13E+72	7.45E+73
25	2.61E+75	9.13E+76
26	3.19E+78	1.12E+80
27	3.91E+81	1.37E+83
28	4.79E+84	1.68E+86
29	5.87E+87	2.06E+89
30	7.19E+90	2.52E+92
31	8.81E+93	3.08E+95
32	1.08E+97	3.78E+98
33	1.32E+100	4.63E+101
34	1.62E+103	5.67E+104
35	1.98E+106	6.95E+107
36	2.43E+109	8.51E+110
37	2.98E+112	1.04E+114
38	3.65E+115	1.28E+117
39	4.47E+118	1.56E+120

Go		
Move	White	Black
1	361	129960
2	46655640	16702719120
3	7.79276E+17	3.63576E+25
4	2.83326E+43	2.20789E+61
5	6.2555E+104	1.7724E+148
6	1.1087E+253	#NUM!
7		
8		
9	After the seventh turn, the numbers are too big for Excel to compute.	
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		
33		
34		
35		
36		
37		
38		
39		

Chess Versus Go

Chess		
Move	White	Black
1	20	700
2	24,500	857,500
3	30,012,500	1,050,437,500
4	36,765,312,500	1.29E+12
5	4.50E+13	1.58E+15
6	5.52E+16	1.93E+18
7	6.76E+19	2.37E+21
8	8.28E+22	2.90E+24
9	1.01E+26	3.55E+27
10	1.24E+29	4.35E+30
11	1.52E+32	5.33E+33
12	1.86E+35	6.53E+36
13	2.28E+38	7.99E+39
14	2.80E+41	9.79E+42
15	3.43E+44	1.20E+46
16	4.20E+47	1.47E+49
17	5.14E+50	1.80E+52
18	6.30E+53	2.21E+55
19	7.72E+56	2.70E+58
20	9.45E+59	3.31E+61
21	1.16E+63	4.05E+64
22	1.42E+66	4.97E+67
23	1.74E+69	6.08E+70
24	2.13E+72	7.45E+73
25	2.61E+75	9.13E+76
26	3.19E+78	1.12E+80
27	3.91E+81	1.37E+83
28	4.79E+84	1.68E+86
29	5.87E+87	2.06E+89
30	7.19E+90	2.52E+92
31	8.81E+93	3.08E+95
32	1.08E+97	3.78E+98
33	1.32E+100	4.63E+101
34	1.62E+103	5.67E+104
35	1.98E+106	6.95E+107
36	2.43E+109	8.51E+110
37	2.98E+112	1.04E+114
38	3.65E+115	1.28E+117
39	4.47E+118	1.56E+120

Go		
Move	White	Black
1	361	129960
2	46655640	16702719120
3	7.79276E+17	3.63576E+25
4	2.83326E+43	2.20789E+61
5	6.2555E+104	1.7724E+148
6	1.1087E+253	#NUM!
7		
8		
9	After the seventh turn, the numbers are too big for Excel to compute.	
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		
33		
34		
35		
36		
37		
38		
39		

Much worse!

Formally Describing the Growth of Functions

Proof of
Factorial vs. Exponential Dominance

Prove the following problem by The Principle of Mathematical Induction:

Problem: For every $n \geq 4$, $n! > 2^n$.

Proof of
Factorial vs. Exponential Dominance

Problem: For every $n \geq 4$, $n! > 2^n$.

Proof:
In this problem $n_0 = 4$.

Basis Step: If $n = 4$, then $LHS = 4! = 24$, and $RHS = 2^4 = 16$.
Hence $LHS > RHS$.
Induction: Assume that $n! > 2^n$ for an arbitrary $n \geq 4$. -- Induction Hypothesis

To prove that this inequality holds for n+1, first try to express *LHS* for n+1 in terms of *LHS* for n and try to use the induction hypothesis.
Note here $(n + 1)! = (n + 1)\, n!$.
Thus using the induction hypothesis, we get $(n + 1)! = (n+1)n! > (n+1)2^n$.

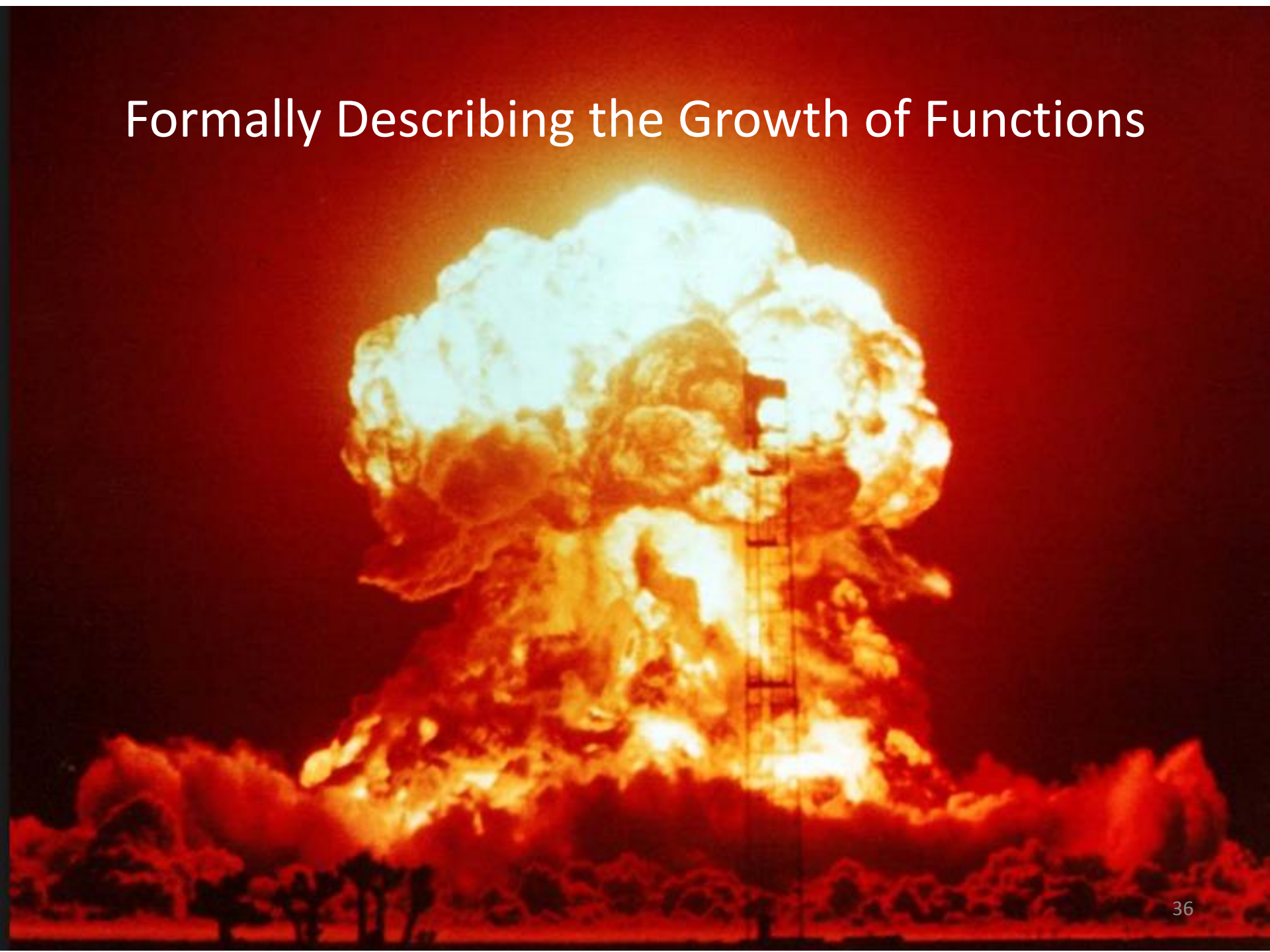
Since $n \geq 4$, $(n+1) > 2$.

Hence $(n+1)2^n > 2^{(n+1)}$.

Hence $(n+1)! > 2^{(n+1)}$.

End of Proof.

Formally Describing the Growth of Functions



Formally Describing the Growth of Functions

Big-O Notation

The growth of functions is often described using a special notation.

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

Example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Solution: We observe that we can readily estimate the size of $f(x)$ when $x > 1$ because $x < x^2$ and $1 < x^2$ when $x > 1$. It follows that

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

whenever $x > 1$, as shown in Figure 1.

Example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Solution: We observe that we can readily estimate the size of $f(x)$ when $x > 1$ because $x < x^2$ and $1 < x^2$ when $x > 1$. It follows that

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

whenever $x > 1$, as shown in Figure 1.

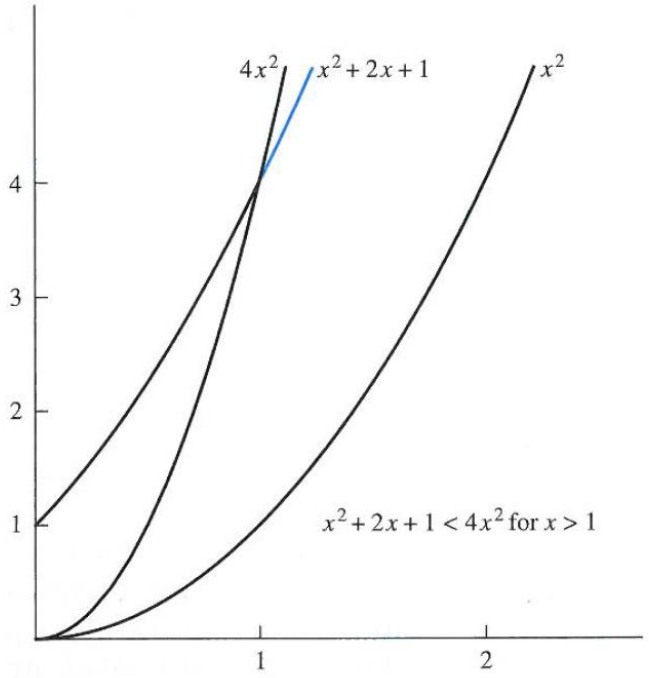


FIGURE 1 The Function $x^2 + 2x + 1$ is $O(x^2)$.

Example

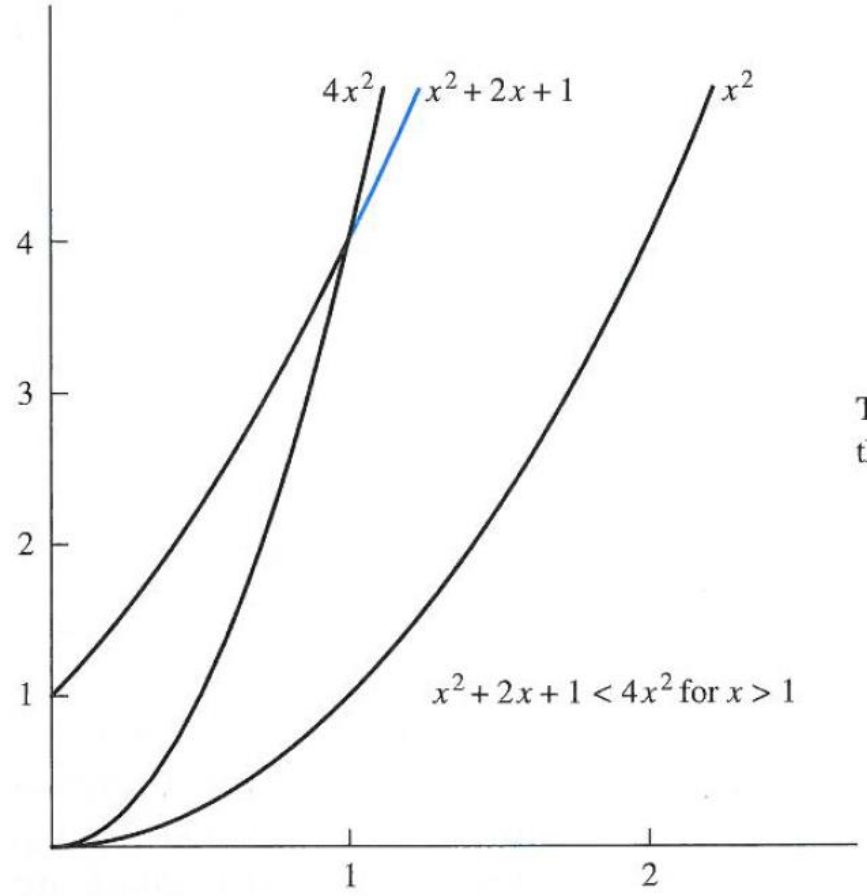
Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Solution: We observe that we can readily estimate the size of $f(x)$ when $x > 1$ because $x < x^2$ and $1 < x^2$ when $x > 1$. It follows that

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

whenever $x > 1$, as shown in Figure 1. Consequently, we can take $C = 4$ and $k = 1$ as witnesses to show that $f(x)$ is $O(x^2)$. That is, $f(x) = x^2 + 2x + 1 < 4x^2$ whenever $x > 1$. (Note that it is not necessary to use absolute values here because all functions in these equalities are positive when x is positive.)

Example



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

FIGURE 1 **The Function $x^2 + 2x + 1$ is $O(x^2)$.**

Example

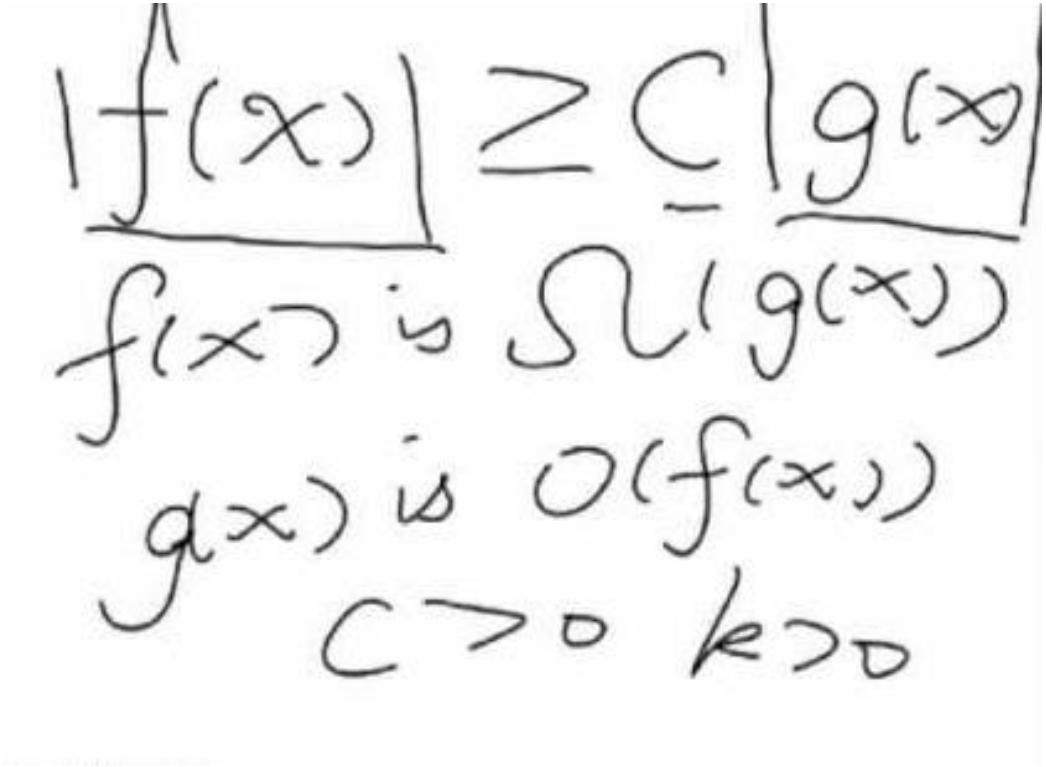
Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Alternatively, we can estimate the size of $f(x)$ when $x > 2$. When $x > 2$, we have $2x \leq x^2$ and $1 \leq x^2$. Consequently, if $x > 2$, we have

$$0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2.$$

It follows that $C = 3$ and $k = 2$ are also witnesses to the relation $f(x)$ is $O(x^2)$.

Big-O, Little-O, Big-Theta, and Big-Omega Notation



Big-O, Little-O, Big-Theta, and Big-Omega Notation

Big-O, Little-o, Omega, and Theta are formal notational methods for stating the growth of resource needs (efficiency and storage) of an algorithm. There are four basic notations used when describing resource needs. These are: O(f(n)), o(f(n)), $\Omega(f(n))$, and $\Theta(f(n))$. (Pronounced, Big-O, Little-O, Omega and Theta respectively)

https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/analysis/notations.html

Big-O, Little-O, Big-Theta, and Big-Omega Notation

Big-O, Little-o, Omega, and Theta are formal notational methods for stating the growth of resource needs (efficiency and storage) of an algorithm. There are four basic notations used when describing resource needs. These are: O(f(n)), o(f(n)), $\Omega(f(n))$, and $\Theta(f(n))$. (Pronounced, Big-O, Little-O, Omega and Theta respectively)

Formally:

"$T(n)$ is $O(f(n))$" iff for some constants c and n_0, $T(n) <= cf(n)$ for all $n >= n_0$

"$T(n)$ is $\Omega(f(n))$" if for some constants c and n_0, $T(n) >= cf(n)$ for all $n >= n_0$

"$T(n)$ is $\Theta(f(n))$" if $T(n)$ is $O(f(n))$ AND $T(n)$ is $\Omega(f(n))$

"$T(n)$ is $o(f(n))$" if $T(n)$ is $O(f(n))$ AND $T(n)$ is NOT $\Theta(f(n))$

https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/analysis/notations.html

Big-O, Little-O, Big-Theta, and Big-Omega Notation

Informally:

"$T(n)$ is $O(f(n))$" basically means that $f(n)$ describes the upper bound for $T(n)$

"$T(n)$ is $\Omega(f(n))$" basically means that $f(n)$ describes the lower bound for $T(n)$

"$T(n)$ is $\Theta(f(n))$" basically means that $f(n)$ describes the exact bound for $T(n)$

"$T(n)$ is $o(f(n))$" basically means that $f(n)$ is the upper bound for $T(n)$ but that $T(n)$ can never be equal to $f(n)$

Another way of saying this:

"$T(n)$ is $O(f(n))$" growth rate of $T(n)$ <= growth rate of $f(n)$

"$T(n)$ is $\Omega(f(n))$" growth rate of $T(n)$ >= growth rate of $f(n)$

"$T(n)$ is $\Theta(f(n))$" growth rate of $T(n)$ == growth rate of $f(n)$

"$T(n)$ is $o(f(n))$" growth rate of $T(n)$ < growth rate of $f(n)$

https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/analysis/notations.html

Algorithmic Classifications

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Analysis of Algorithms

- In computing science, the **analysis of algorithms** is the determination of the amount of resources (such as time and storage) necessary to execute them.

Analysis of Algorithms

- In computing science, the **analysis of algorithms** is the determination of the amount of resources (such as time and storage) necessary to execute them.
- Most algorithms are designed to work with inputs of arbitrary length.

Analysis of Algorithms

- In computing science, the **analysis of algorithms** is the determination of the amount of resources (such as time and storage) necessary to execute them.
- Most algorithms are designed to work with inputs of arbitrary length.
- Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (**time complexity**) or storage locations (**space complexity**).

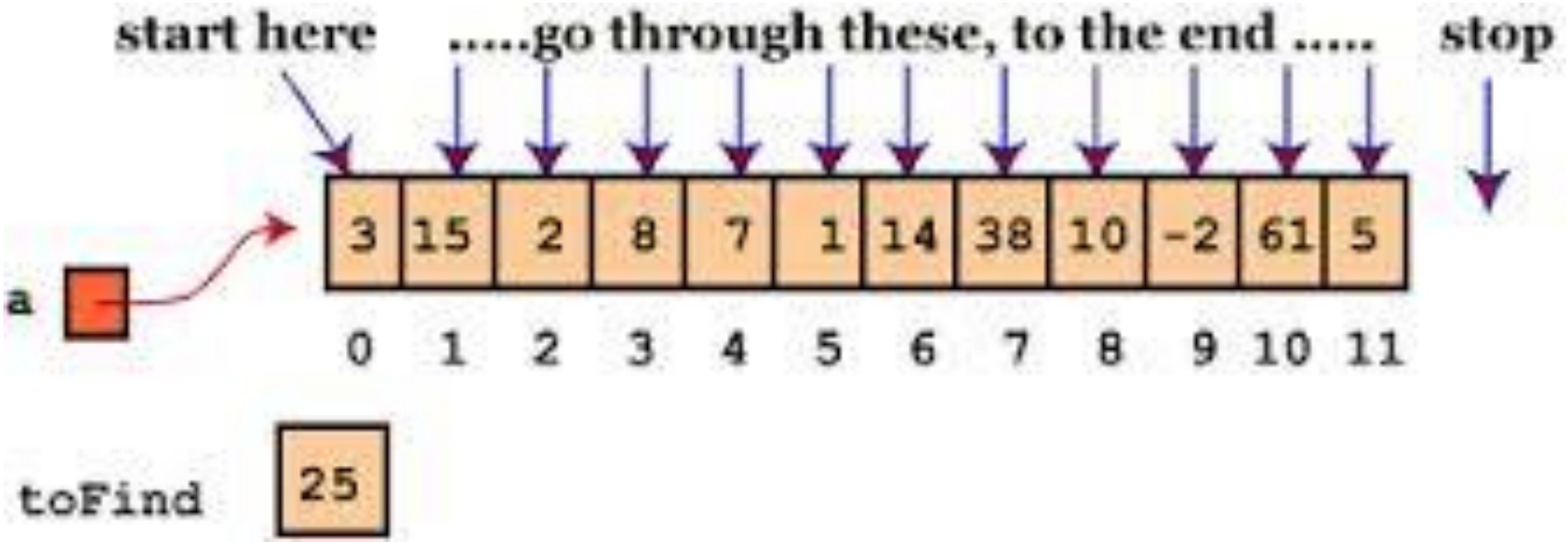
Best, Worst, Average Case

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size *n* may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size *n*.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size *n*.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size *n*.

Wikipedia

Best, Worst, Average Case



Best Case:	First element.
Average Case:	n/2.
Worst Case:	Last element or not present (**as here**).

Computational Time of Various Classes of Problems According to Input Size

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*			
n	$\log n$	n	$n \log n$	n^2
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min

Computational Time of Various Classes of Problems According to Input Size

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

Computational Time of Various Classes of Problems According to Input Size

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

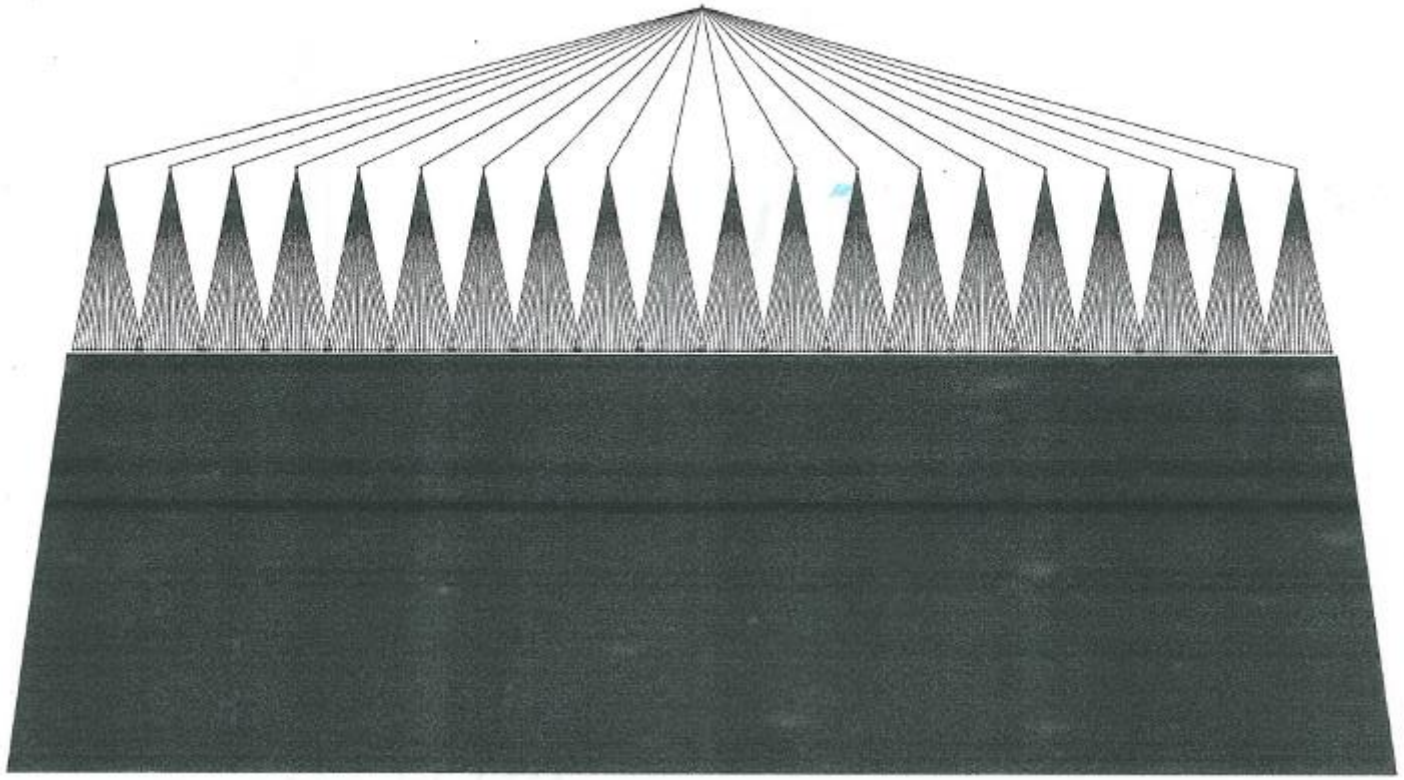
TRACTABLE → ← INTRACTABLE →

Definition

Tractable Problem:

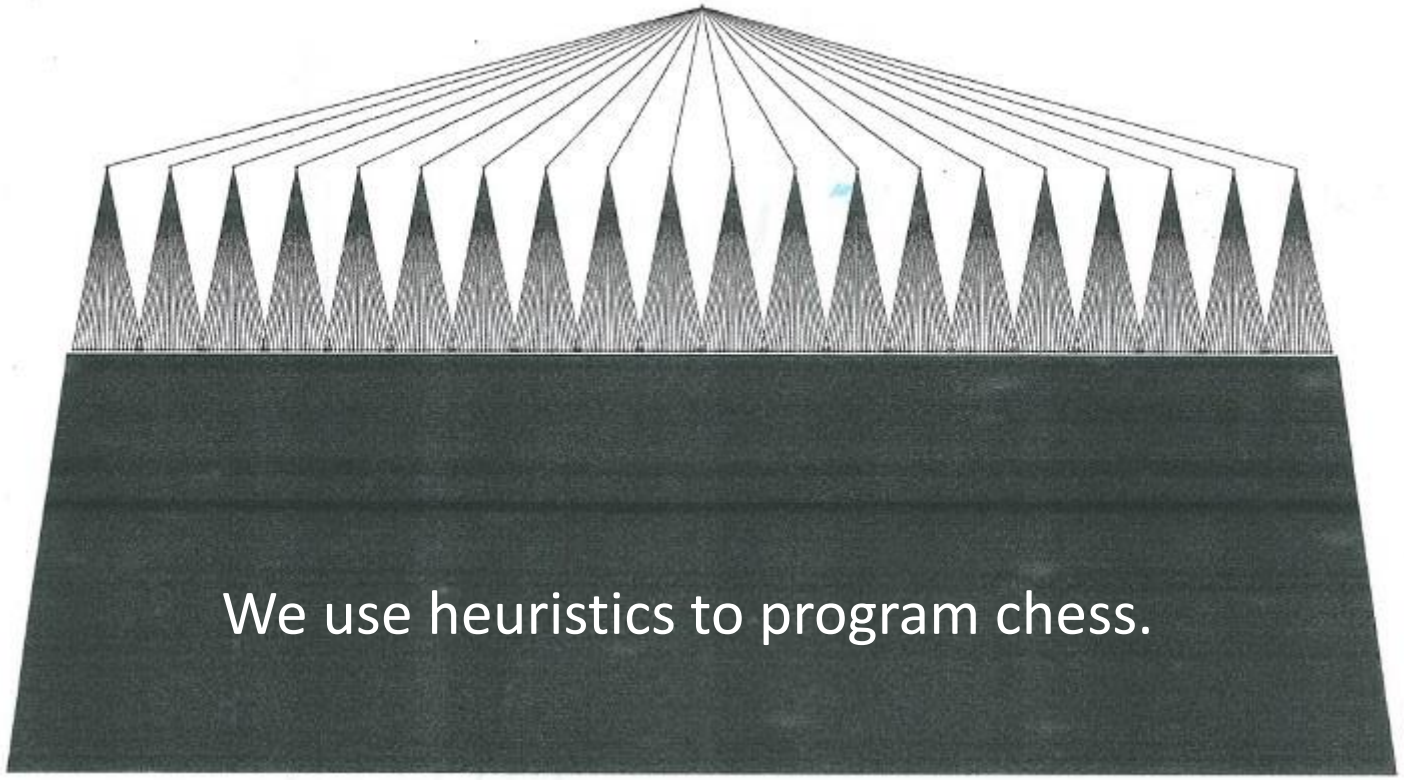
An algorithm that has at most worst-case polynomial time complexity.

The Game of Chess is Intractable



~35^d

The Game of Chess is Intractable



~35^d

What is Logarithmic Complexity?

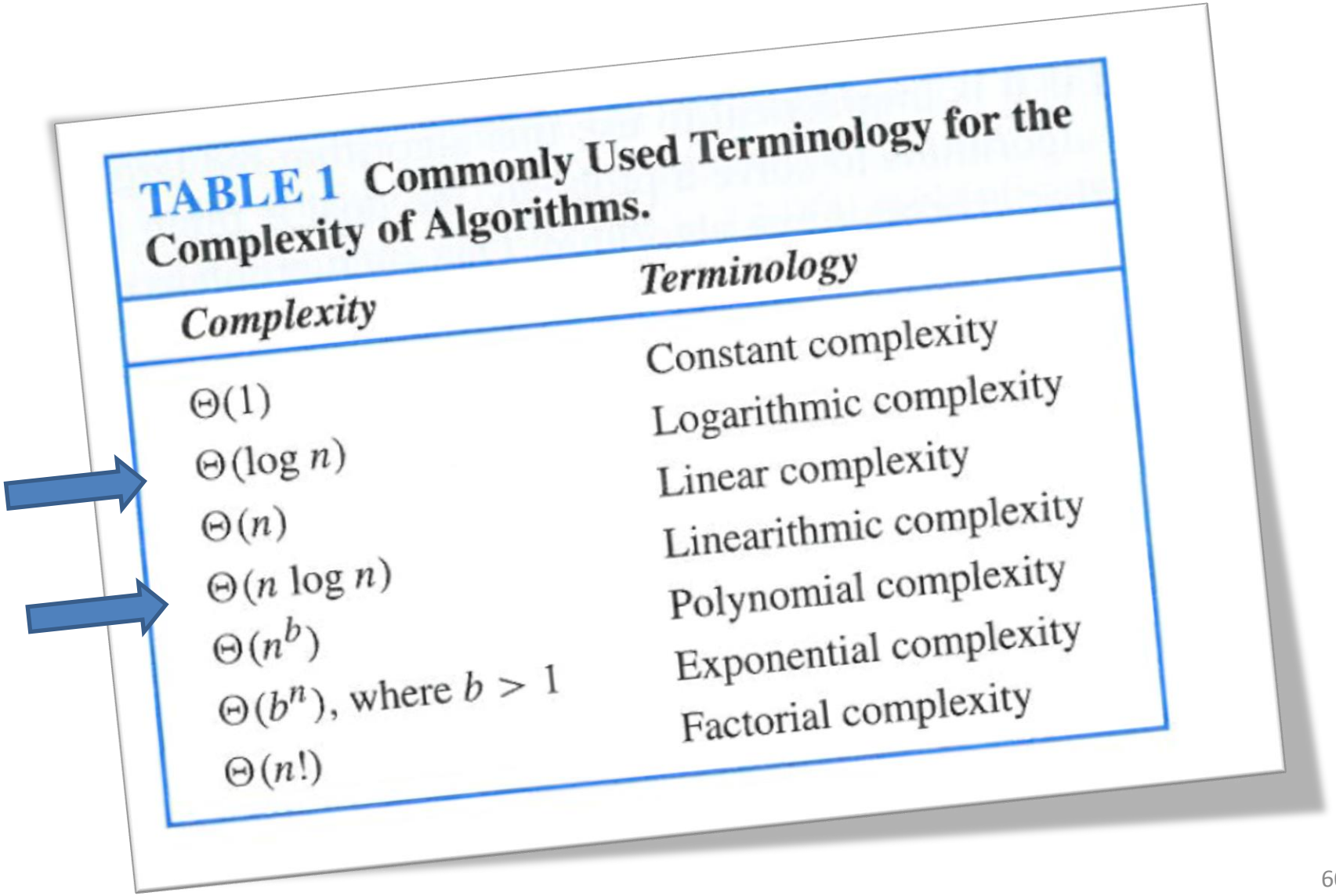


TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

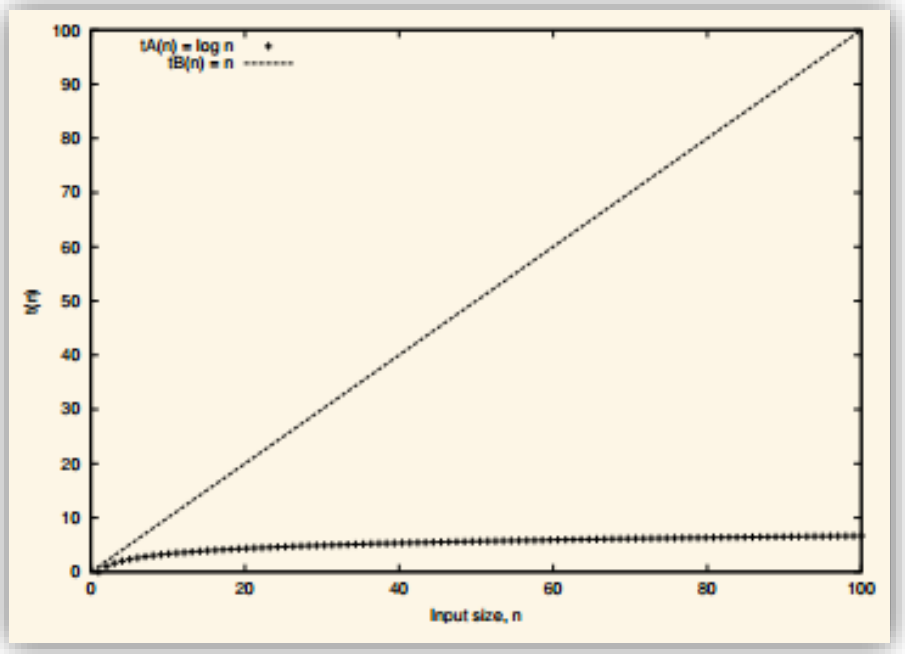
Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

What is Logarithmic Complexity?

- In complexity theory, the complexity functions for algorithms that repeatedly split their input into two halves involve base 2 logs.

What is Logarithmic Complexity?

- In complexity theory, the complexity functions for algorithms that repeatedly split their input into two halves involve base 2 logs.
- Suppose algorithm A's worst-case time complexity $t_A(n) = \log n$ and algorithm B's worst-case time complexity $t_B(n) = n$.
  - log*n* grows much more slowly than *n*.



Classes of Time Complexity

Linear complexity [change | change source]

Complexity theory also looks at how a problem changes if it is done for more elements. Mowing the lawn can be thought of as a problem with linear complexity. Mowing an area that is double the size of the original takes twice as long.

Wikipedia

Classes of Time Complexity

Linear complexity [change | change source]

Complexity theory also looks at how a problem changes if it is done for more elements. Mowing the lawn can be thought of as a problem with linear complexity. Mowing an area that is double the size of the original takes twice as long.

Quadratic complexity [change | change source]

Suppose you want to know which of your friends know each other. You have to ask each pair of friends whether they know each other. If you have **twice** as many friends as someone else, you have to ask **four** times as many questions to figure out who everyone knows. Problems that take four times as long when the size of the problem doubles are said to have **quadratic complexity**.

Wikipedia

Classes of Time Complexity

Linear complexity [change | change source]

Complexity theory also looks at how a problem changes if it is done for more elements. Mowing the lawn can be thought of as a problem with linear complexity. Mowing an area that is double the size of the original takes twice as long.

Quadratic complexity [change | change source]

Suppose you want to know which of your friends know each other. You have to ask each pair of friends whether they know each other. If you have **twice** as many friends as someone else, you have to ask **four** times as many questions to figure out who everyone knows. Problems that take four times as long when the size of the problem doubles are said to have **quadratic complexity**.

Logarithmic complexity [change | change source]

This is often the complexity for problems that involve looking things up, like finding a word in a dictionary. If the dictionary is twice as big, it contains twice as many words as the original to compare to. Looking something up will take only one step more. The algorithm to do lookups is simple. The word in the middle of the dictionary will be either before or after the term that needs to be looked up, if the words do not match. If it is before, the term needs to be in the second half of the dictionary. If it is after the word, it needs to be in the first half. That way, the problem space is halved with every step, until the word or definition is found. This is generally known as **logarithmic complexity**

Wikipedia

Classes of Time Complexity

Linear complexity [change | change source]

Complexity theory also looks at how a problem changes if it is done for more elements. Mowing the lawn can be thought of as a problem with linear complexity. Mowing an area that is double the size of the original takes twice as long.

Quadratic complexity [change | change source]

Suppose you want to know which of your friends know each other. You have to ask each pair of friends whether they know each other. If you have **twice** as many friends as someone else, you have to ask **four** times as many questions to figure out who everyone knows. Problems that take four times as long when the size of the problem doubles are said to have **quadratic complexity**.

Logarithmic complexity [change | change source]

This is often the complexity for problems that involve looking things up, like finding a word in a dictionary. If the dictionary is twice as big, it contains twice as many words as the original to compare to. Looking something up will take only one step more. The algorithm to do lookups is simple. The word in the middle of the dictionary will be either before or after the term that needs to be looked up, if the words do not match. If it is before, the term needs to be in the second half of the dictionary. If it is after the word, it needs to be in the first half. That way, the problem space is halved with every step, until the word or definition is found. This is generally known as **logarithmic complexity**

Exponential complexity [change | change source]

There are problems that grow very fast. One such problem is known as the Travelling salesman problem. A salesman needs to take a tour of a certain number of cities. Each city should only be visited once, the distance (or cost) of the travelling should be minimal, and the salesman should end up where he started. This problem has **exponential complexity**. There are n factorial possibilities to consider. Adding one city (from n to n+1) will multiply the number of possibilities by (n+1). Most of the interesting problems have this complexity.

Wikipedia

Time Complexity of Simple Tractable Problems

Example #1

Calculate the time complexity of Max Search:

ALGORITHM 1 Finding the Maximum Element in a Finite Sequence.

procedure $max(a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i := 2$ **to** n
    if $max < a_i$ **then** $max := a_i$
return max {max is the largest element}

Rosen

Example #1

Calculate the time complexity of Max Search:

Solution: The number of comparisons will be used as the measure of the time complexity of the algorithm, because comparisons are the basic operations used.

Please attempt this problem

Rosen

Example #1

Calculate the time complexity of Max Search:

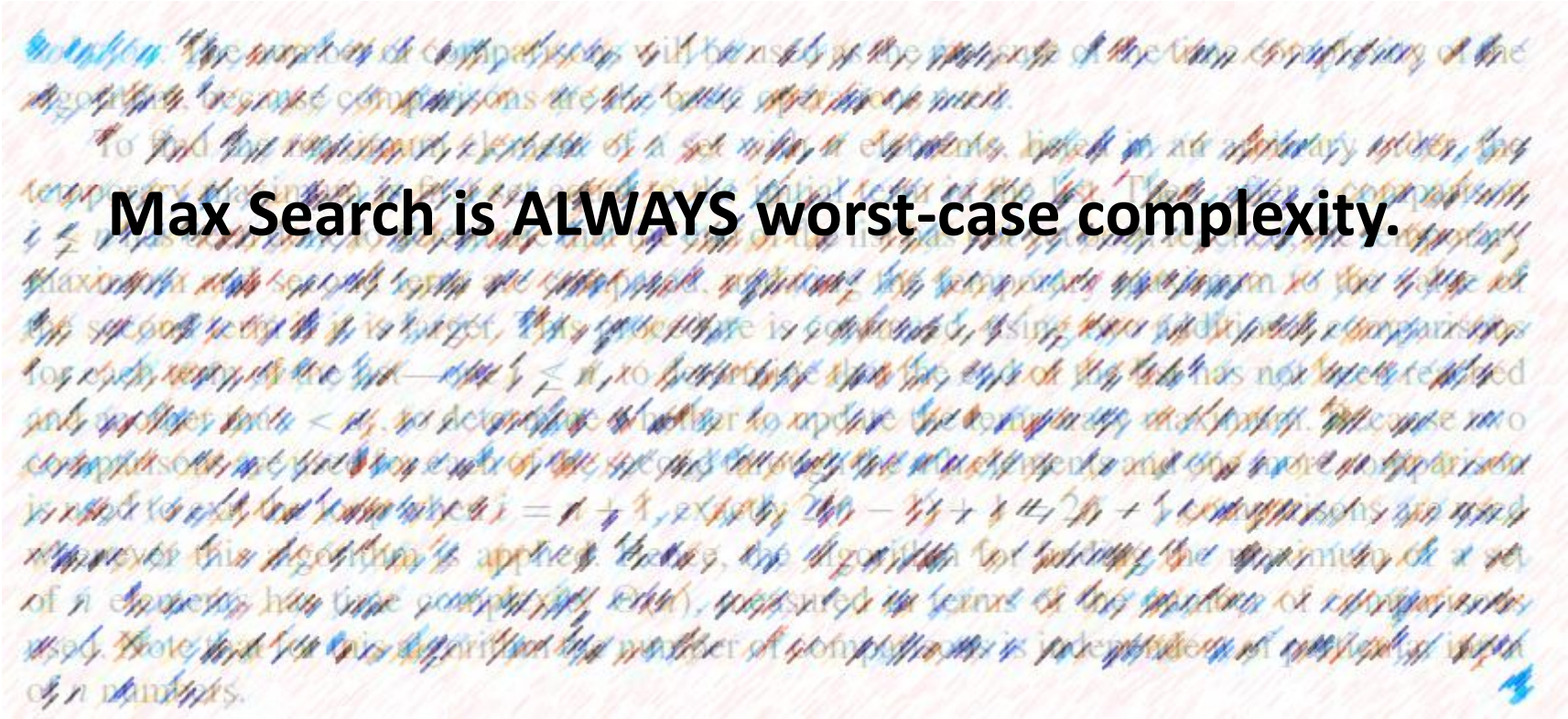
Solution: The number of comparisons will be used as the measure of the time complexity of the algorithm, because comparisons are the basic operations used.

To find the maximum element of a set with n elements, listed in an arbitrary order, the temporary maximum is first set equal to the initial term in the list. Then, after a comparison $i \leq n$ has been done to determine that the end of the list has not yet been reached, the temporary maximum and second term are compared, updating the temporary maximum to the value of the second term if it is larger. This procedure is continued, using two additional comparisons for each term of the list—one $i \leq n$, to determine that the end of the list has not been reached and another max $< a_i$, to determine whether to update the temporary maximum. Because two comparisons are used for each of the second through the nth elements and one more comparison is used to exit the loop when $i = n + 1$, exactly $2(n - 1) + 1 = 2n - 1$ comparisons are used whenever this algorithm is applied. Hence, the algorithm for finding the maximum of a set of n elements has time complexity $\Theta(n)$, measured in terms of the number of comparisons used. Note that for this algorithm the number of comparisons is independent of particular input of n numbers. ◀

Rosen

Example #1

Calculate the time complexity of Max Search:



Example #2

Calculate the time complexity of Linear Search:

ALGORITHM 2 The Linear Search Algorithm.

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
    $i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
return *location*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

Rosen

Example #2

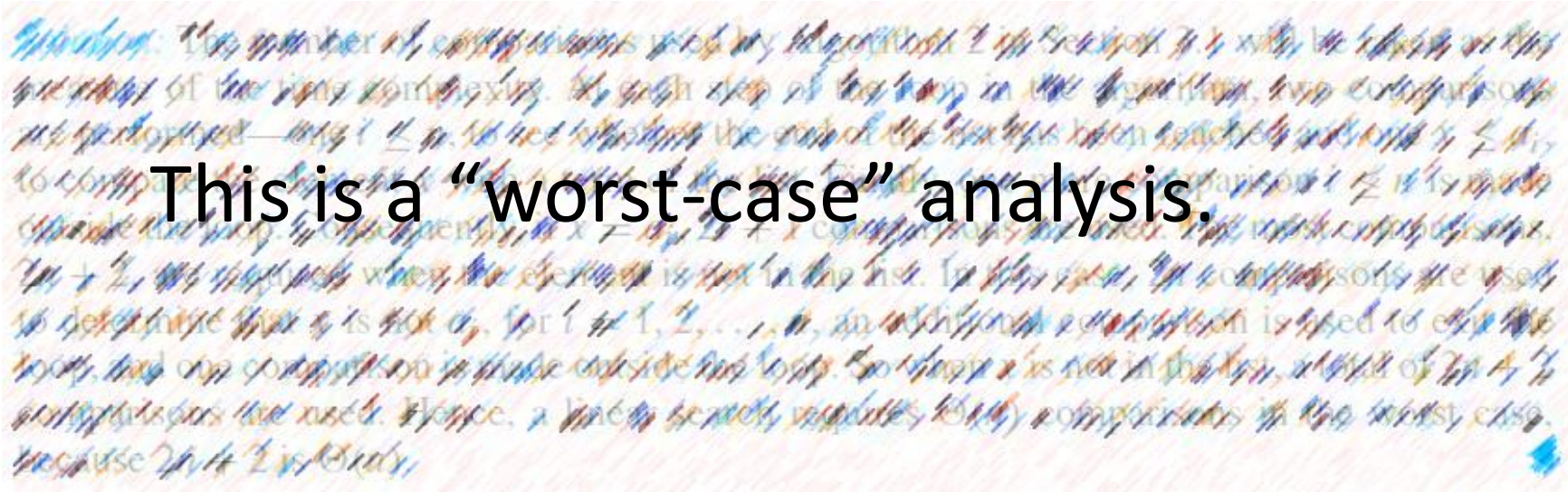
Calculate the time complexity of Linear Search:

Solution: The number of comparisons used by Algorithm 2 in Section 3.1 will be taken as the measure of the time complexity. At each step of the loop in the algorithm, two comparisons are performed—one $i \leq n$, to see whether the end of the list has been reached and one $x \leq a_i$, to compare the element x with a term of the list. Finally, one more comparison $i \leq n$ is made outside the loop. Consequently, if $x = a_i$, $2i + 1$ comparisons are used. The most comparisons, $2n + 2$, are required when the element is not in the list. In this case, $2n$ comparisons are used to determine that x is not a_i, for $i = 1, 2, \ldots, n$, an additional comparison is used to exit the loop, and one comparison is made outside the loop. So when x is not in the list, a total of $2n + 2$ comparisons are used. Hence, a linear search requires $\Theta(n)$ comparisons in the worst case, because $2n + 2$ is $\Theta(n)$. ◀

Rosen

Example #2

Calculate the time complexity of Linear Search:



Example #3

Determine the time complexity of matrix scalar multiplication, which is defined as follows:

The **scalar product** of a real number, r, and a matrix A is the matrix rA. Each element of matrix rA is r times its corresponding element in A.

Given scalar r and matrix $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $rA = \begin{bmatrix} ra_{11} & ra_{12} \\ ra_{21} & ra_{22} \end{bmatrix}$.

Example #3

Determine the time complexity of matrix scalar multiplication, which is defined as follows:

The **scalar product** of a real number, r, and a matrix A is the matrix rA. Each element of matrix rA is r times its corresponding element in A.

Given scalar r and matrix $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $rA = \begin{bmatrix} ra_{11} & ra_{12} \\ ra_{21} & ra_{22} \end{bmatrix}$.

Answer: $O(n^2)$ for an nxn matrix

Example #4

Determine the time complexity of matrix multiplication, which is defined for *nxn* matrices *as* follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

i.e., $$\mathbf{AB} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\lambda + c\rho & a\beta + b\mu + c\sigma & a\gamma + b\nu + c\tau \\ p\alpha + q\lambda + r\rho & p\beta + q\mu + r\sigma & p\gamma + q\nu + r\tau \\ u\alpha + v\lambda + w\rho & u\beta + v\mu + w\sigma & u\gamma + v\nu + w\tau \end{pmatrix},$$

Example #4

Determine the time complexity of matrix multiplication, which is defined for *nxn* matrices *as* follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

i.e., $$\mathbf{AB} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\lambda + c\rho & a\beta + b\mu + c\sigma & a\gamma + b\nu + c\tau \\ p\alpha + q\lambda + r\rho & p\beta + q\mu + r\sigma & p\gamma + q\nu + r\tau \\ u\alpha + v\lambda + w\rho & u\beta + v\mu + w\sigma & u\gamma + v\nu + w\tau \end{pmatrix},$$

ALGORITHM 1 Matrix Multiplication.

procedure *matrix multiplication*(**A**, **B**: matrices)
for $i := 1$ **to** m
  for $j := 1$ **to** n
    $c_{ij} := 0$
    for $q := 1$ **to** k
      $c_{ij} := c_{ij} + a_{iq} b_{qj}$
return **C** {**C** = $[c_{ij}]$ is the product of **A** and **B**}

Example #5

Compare the time complexities of the following:

ALGORITHM 5 The Insertion Sort.

procedure *insertion sort*($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)
for $j := 2$ **to** n
    $i := 1$
    while $a_j > a_i$
        $i := i + 1$
    $m := a_j$
    for $k := 0$ **to** $j - i - 1$
        $a_{j-k} := a_{j-k-1}$
    $a_i := m$
{$a_1, \ldots, a_n$ is in increasing order}

ALGORITHM 4 The Bubble Sort.

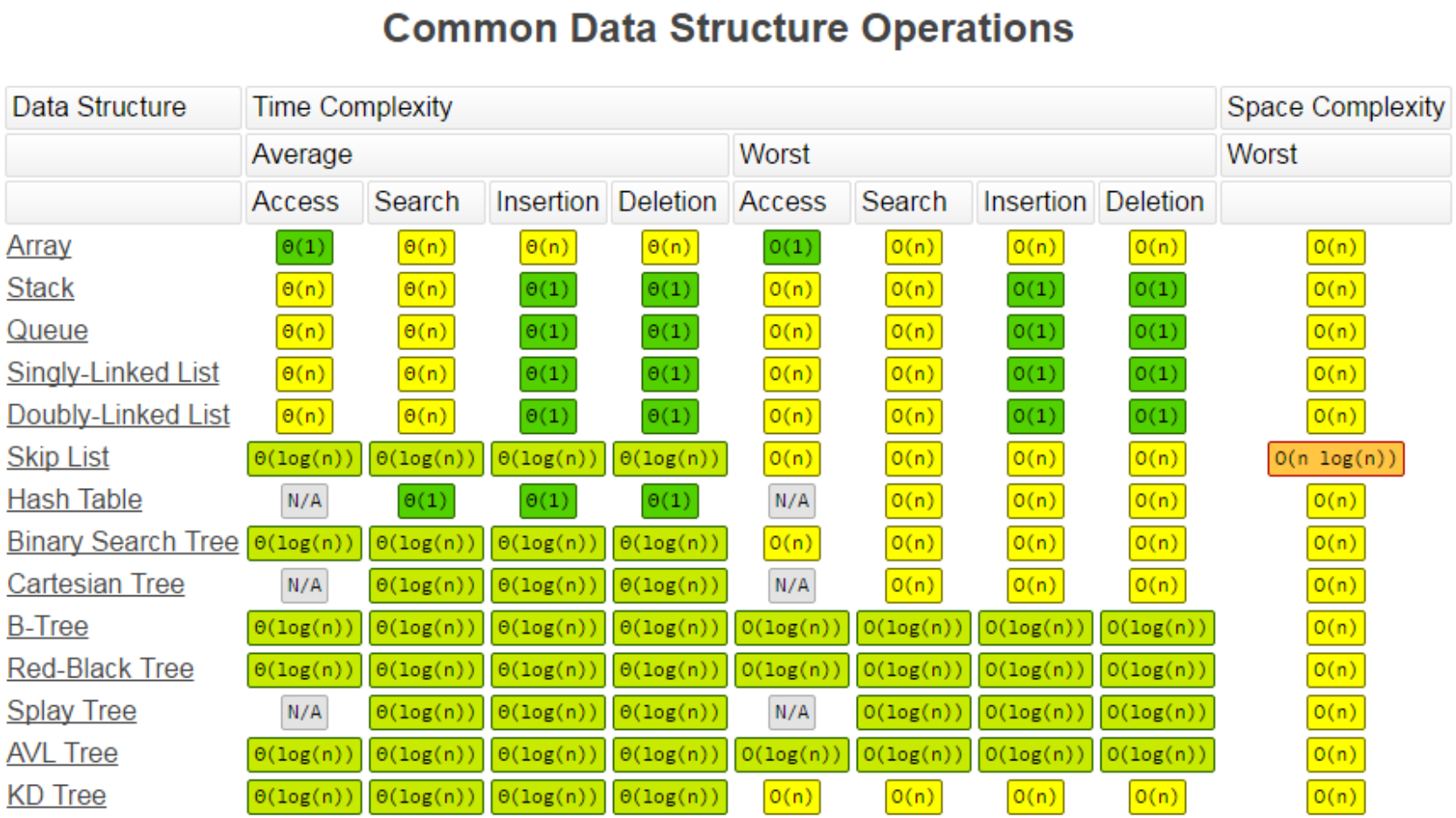
procedure *bubblesort*($a_1, \ldots, a_n$: real numbers with $n \geq 2$)
for $i := 1$ **to** $n - 1$
    for $j := 1$ **to** $n - i$
        if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}
{$a_1, \ldots, a_n$ is in increasing order}

Exercise

In these two examples we showed that both the bubble sort and the insertion sort have worst-case time complexity (n^2).

However, the most efficient sorting algorithms can sort *n* items in *O(n* log *n)* time.

Time Complexity Summary

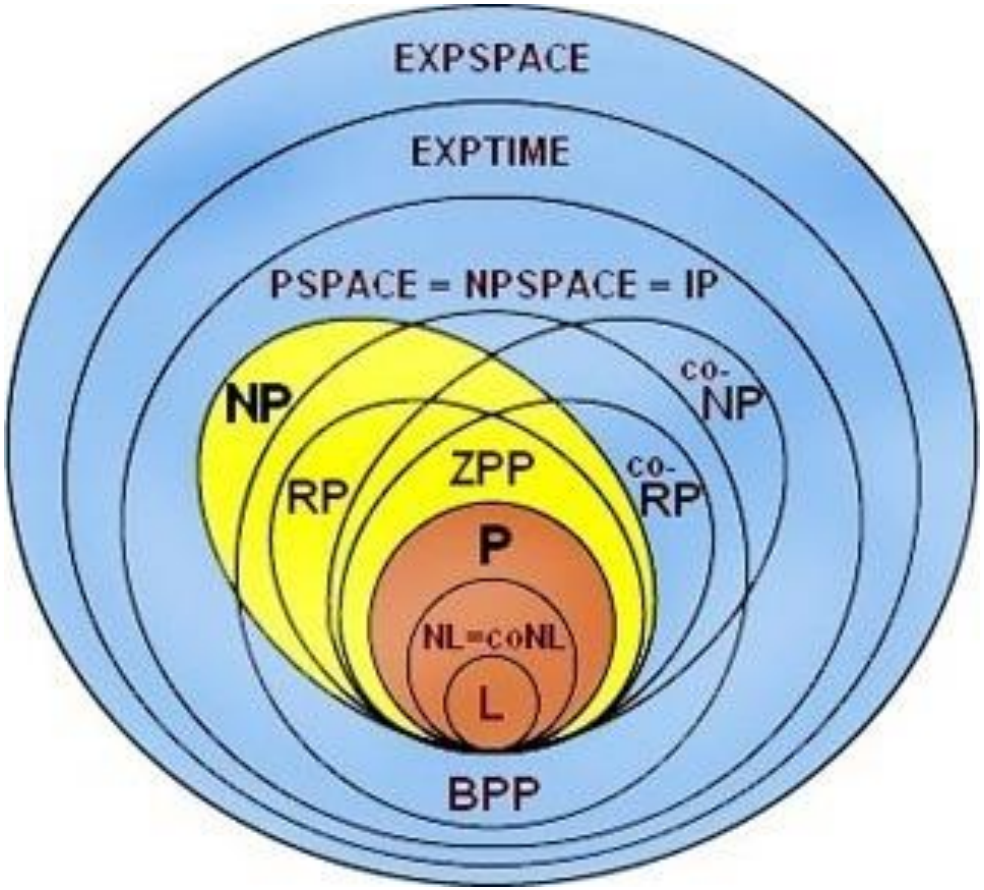


Common Data Structure Operations

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

http://bigocheatsheet.com/

APPENDIX:


EXPSPACE
EXPTIME
PSPACE = NPSPACE = IP
NP
co-NP
RP
ZPP
co-RP
P
NL=coNL
L
BPP

Advanced Computational Complexity Theory

- The branch of theory in theoretical computing science and mathematics that deals with classifying problems according to their inherent difficulty and their relationship to one another.

Advanced Computational Complexity Theory

- The branch of theory in theoretical computing science and mathematics that deals with classifying problems according to their inherent difficulty and their relationship to one another.
- A computational problem is regarded as a task (theoretically, an infinite collection of "**problem instances**" and their individual solutions) that can be solved by mechanically applied mathematical steps, say, by a person or machine (executing an algorithm).

Advanced Computational Complexity Theory

- The size of a problem instance is central to this study.

Advanced Computational Complexity Theory

- The size of a problem instance is central to this study.
- To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem.

Advanced Computational Complexity Theory

- The size of a problem instance is central to this study.
- To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem.
- However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve.

Advanced Computational Complexity Theory

- The size of a problem instance is central to this study.
- To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem.
- However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve.
- Thus the time required to solve a problem (or the space required, or any measure of complexity) is calculated as a function of the size of the instance.

Advanced Computational Complexity Theory

- The size of a problem instance is central to this study.
- To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem.
- However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve.
- Thus the time required to solve a problem (or the space required, or any measure of complexity) is calculated as a function of the size of the instance.
- This is usually taken to be the size of the input in bits. *Complexity theory is interested in how algorithms scale with an increase in the input size*, as we have seen.

Advanced Computational Complexity Theory

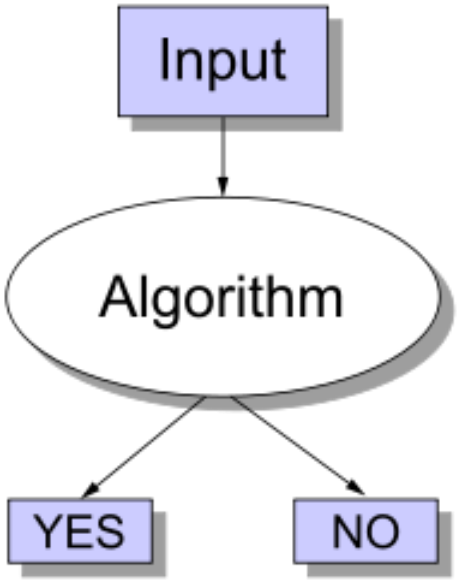
Prerequisites:

- Problem Type: **Simple Decision Problem**
- Model of a “Computer”: **Turing Machine** (2 types)
  - **Deterministic** (the regular computer we always thinking of)
  - N**on-deterministic** (just like the one we're used to except that is has unlimited *parallelism*, so that any time you come to a branch, you spawn a new "process" and examine both sides).

Advanced Computational Complexity Theory

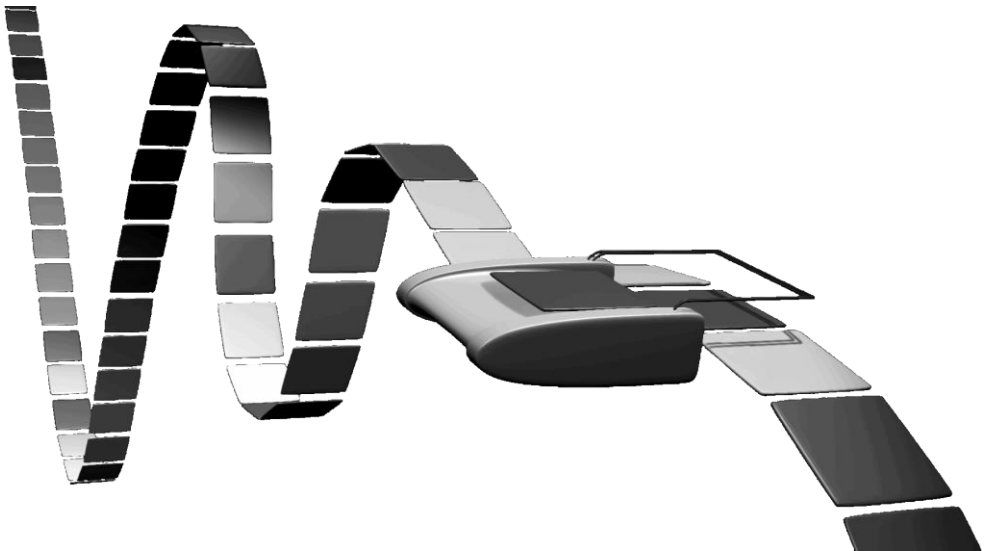
Definition: Decision Problem

A problem for which an algorithm can always answer "yes" or "no."



Machine Model: The Turing Machine

A Turing machine is a mathematical model of a general computing machine. It is a theoretical device that manipulates symbols contained on a strip of tape. Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine—anything from an advanced supercomputer to a mathematician with a pencil and paper.



Wikipedia

Definition: Class P

Class P: A decision problem which increases in time as a polynomial function of the problem size, n.

- *Example #1:* Linear Search, $O(n)$.
- *Example #2:* Matrix Multiplication, $O(n^3)$.

Definition: Class NP

Class NP: Decision problems solvable in time which increases faster than polynomial in *n*, but once the solution is obtained if one exists, it can be **verified** in polynomial time in *n*.

- *Example:* Integer factorization problem, i.e., decomposing an integer into two factors.

Definition: Class NP

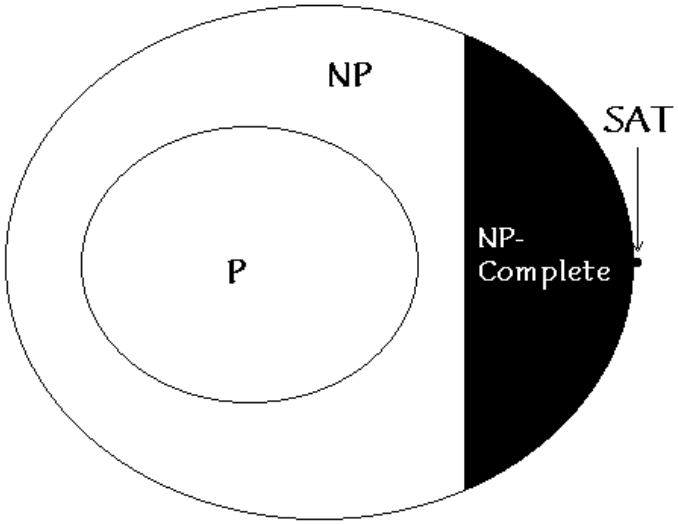
Class NP: Decision problems solvable in time which increases faster than polynomial in *n*, but once the solution is obtained if one exists, it can be **verified** in polynomial time in *n*.

- *Example:* Integer factorization problem, i.e., decomposing an integer into two factors.

In other words, A decision problem is in NP if there is a known polynomial-time algorithm for a **non-deterministic machine** to get the answer.

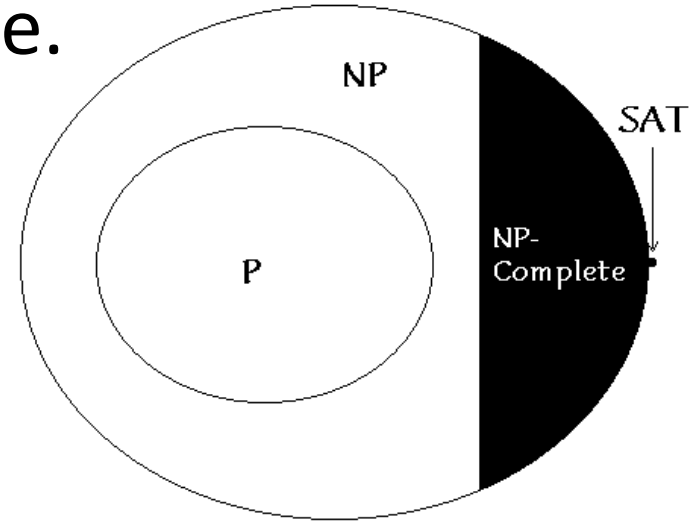
NOTE

- Problems known to be in P are trivially in NP.



NOTE

- Problems known to be in P are trivially in NP.
- The nondeterministic machine just never troubles itself to fork another process, and acts just like a deterministic one and always makes the correct choice.

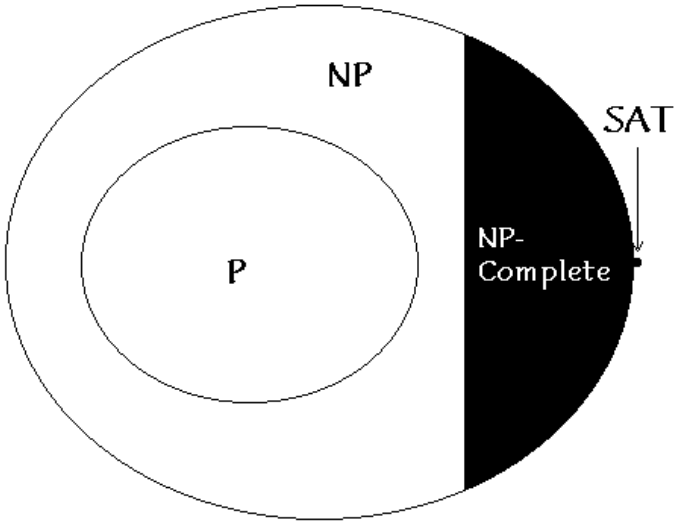


NOTE

- Problems known to be in P are trivially in NP.
- The nondeterministic machine just never troubles itself to fork another process, and acts just like a deterministic one and always makes the correct choice.
- However, there are problems that are known to be neither in P nor NP; a simple example is the game of chess.

Definition: Class NP Complete

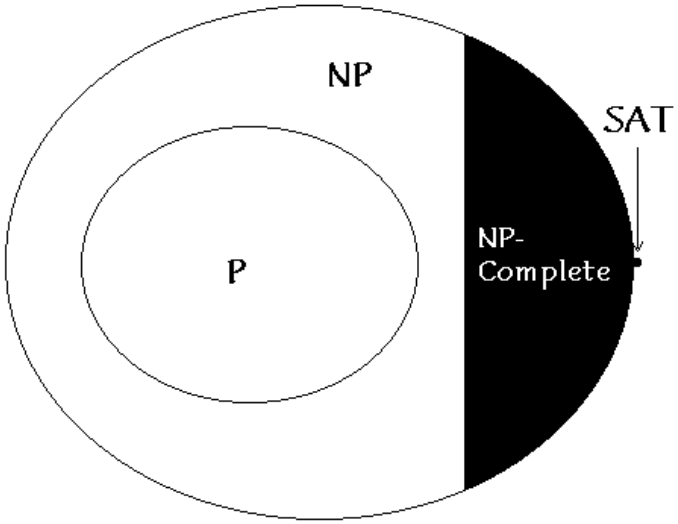
Class NP-complete: Decision problems in NP, to which all other problems in NP can be reduced in polynomial time.



Definition: Class NP Complete

Class NP-complete: Decision problems in NP, to which all other problems in NP can be reduced in polynomial time.

- An example of an NP-complete problem is ``satisfiability'' (SAT): finding truth values for *n* variables which make a particular Boolean expression true.

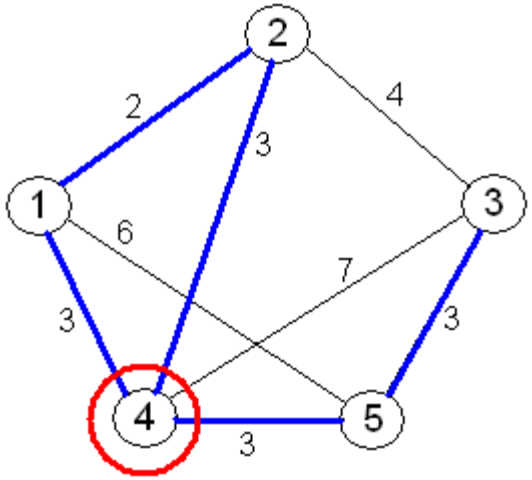


Class NP Complete

- These are problems which are known to be in NP for which no polynomial-time deterministic algorithm is known; in other words, we know they're in NP, but don't know if they're in P.

Class NP Complete

- These are problems which are known to be in NP for which no polynomial-time deterministic algorithm is known; in other words, we know they're in NP, but don't know if they're in P.
- The traditional example is the decision-problem version of the **Traveling Salesman Problem** (decision-TSP): given the cities and distances, is there a route that covers all the cities, returning to the starting point, in less than *x*?

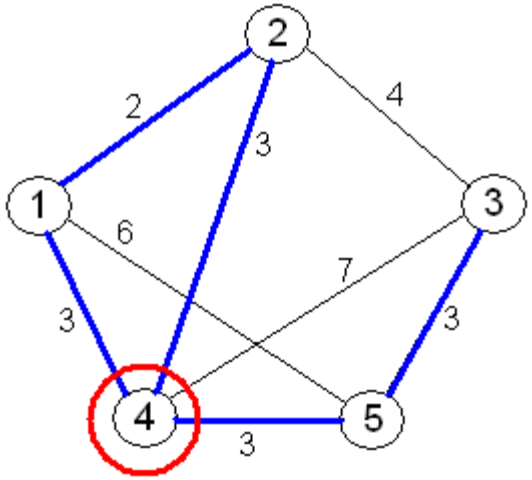


Class NP Complete

- These are problems which are known to be in NP for which no polynomial-time deterministic algorithm is known; in other words, we know they're in NP, but don't know if they're in P.
- The traditional example is the decision-problem version of the **Traveling Salesman Problem** (decision-TSP): given the cities and distances, is there a route that covers all the cities, returning to the starting point, in less than *x*?
- It's easy in a nondeterministic machine, because every time the nondeterministic traveling salesman comes to a fork in the road, he takes it: his clones head on to the next city they haven't visited, and at the end they compare notes and see if any of the clones took less than *x*.

Travelling Salesman Problem: TSP

It's not known whether Decision-TSP is in P: there's no known polynomial-time solution, but there's no proof such a solution doesn't exist.



Class NP, NP-Complete and NP-Hard

- A problem is NP-complete if you can prove that
  1. it's in NP, and
  2. show that it's poly-time reducible to a problem already known to be NP-complete.

http://stackoverflow.com/

Class NP, NP-Complete and NP-Hard

- A problem is NP-complete if you can prove that
  1. it's in NP, and
  2. show that it's poly-time reducible to a problem already known to be NP-complete
- So really, what it says is that if anyone ever finds a poly-time solution to one NP-complete problem, they've automatically got one for *all* the NP-complete problems; that will also mean that **P=NP (THE GREATEST UNSOLVED PROBLEM OF COMPUTING SCIENCE)**.

http://stackoverflow.com/

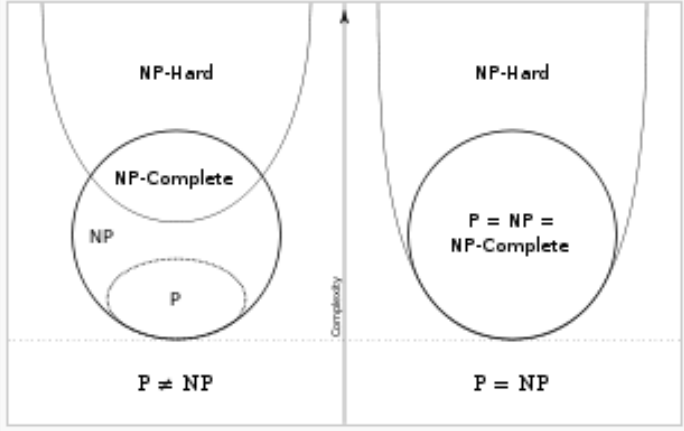
Class NP, NP-Complete and NP-Hard

- A problem is NP-complete if you can prove that
  1. it's in NP, and
  2. show that it's poly-time reducible to a problem already known to be NP-complete.
- So really, what it says is that if anyone ever finds a poly-time solution to one NP-complete problem, they've automatically got one for *all* the NP-complete problems; that will also mean that P=NP **(THE GREATEST UNSOLVED PROBLEM OF COMPUTING SCIENCE)**.
- **NOTE**: A problem is **NP-hard** *iff* it is *at least as hard as* an NP-complete problem. The more conventional Traveling Salesman Problem (not the Decision TSP) of finding the shortest route is NP-hard, not strictly NP-complete.

http://stackoverflow.com/

Definition: NP-Hard

NP-hard (**N**on-deterministic **P**olynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". A problem *H* is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H (i.e., $L \leq_T H$). In other words, *L* can be solved in polynomial time by an oracle machine with an oracle for *H*. Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving *H*, and solves *L* in polynomial time, if the subroutine call takes only one step to compute. NP-hard problems may be of any type: decision problems, search problems, or optimization problems.



Euler diagram for P, NP, NP-complete, and NP-hard set of problems

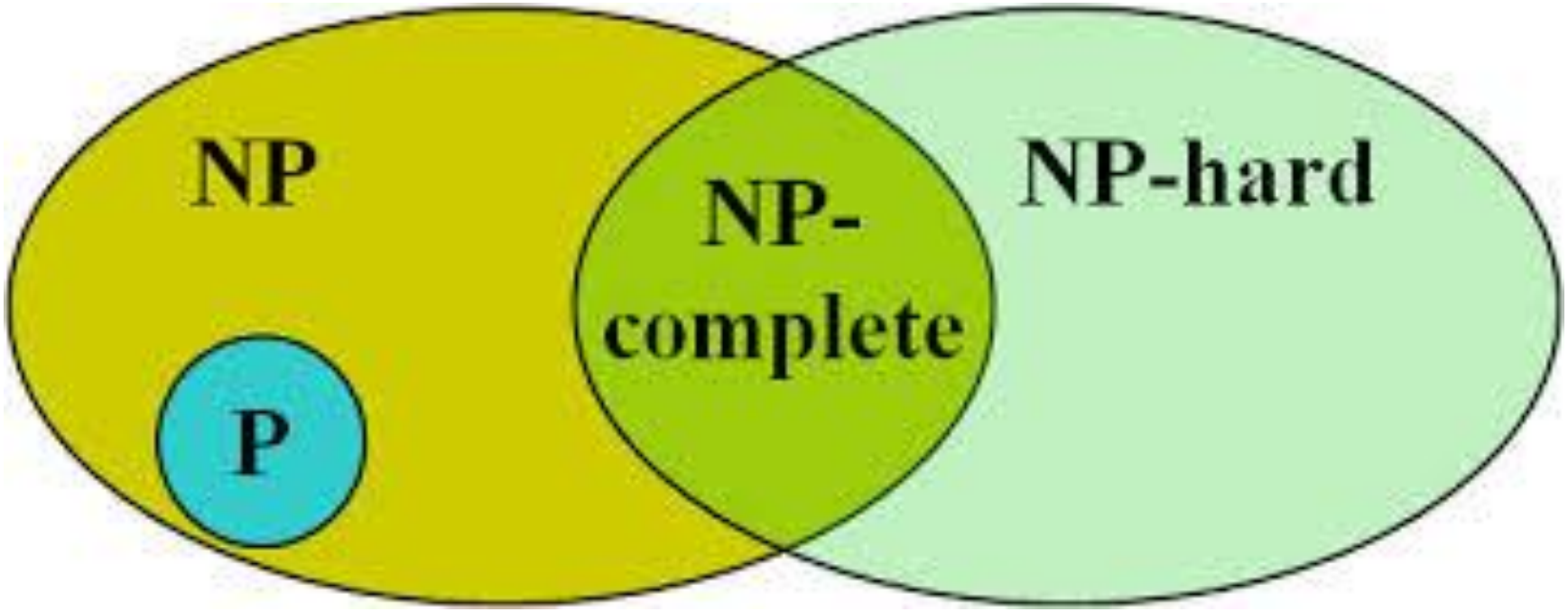
P = NP?

- If any problem in NP can be *decided* in polynomial time, then ALL problems in NP can be *solved* in polynomial time.

P = NP?

- If any problem in NP can be *decided* in polynomial time, then ALL problems in NP can be *solved* in polynomial time.
- Thus, it would be shown that the class P of problems deterministically decidable in polynomial time would equal the class NP of problems non-deterministically decidable in polynomial time.

Summary Diagram

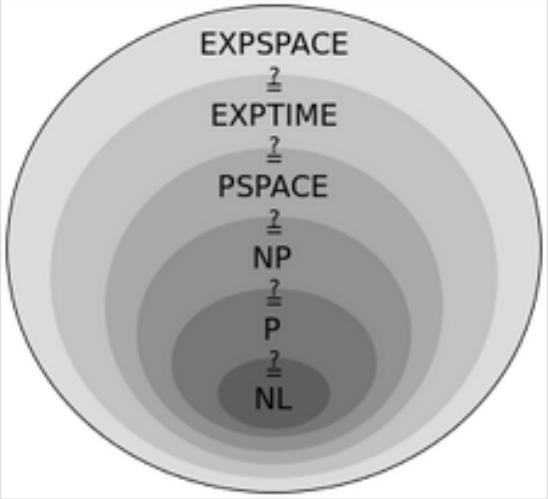


In-Depth Summary

Important complexity classes [edit]

Many important complexity classes can be defined by bounding the time or space used by the algorithm. Some important complexity classes of decision problems defined in this manner are the following:

Complexity class	Model of computation	Resource constraint
DTIME($f(n)$)	Deterministic Turing machine	Time $f(n)$
P	Deterministic Turing machine	Time poly(n)
EXPTIME	Deterministic Turing machine	Time $2^{\text{poly}(n)}$
NTIME($f(n)$)	Non-deterministic Turing machine	Time $f(n)$
NP	Non-deterministic Turing machine	Time poly(n)
NEXPTIME	Non-deterministic Turing machine	Time $2^{\text{poly}(n)}$
DSPACE($f(n)$)	Deterministic Turing machine	Space $f(n)$
L	Deterministic Turing machine	Space O(log n)
PSPACE	Deterministic Turing machine	Space poly(n)
EXPSPACE	Deterministic Turing machine	Space $2^{\text{poly}(n)}$
NSPACE($f(n)$)	Non-deterministic Turing machine	Space $f(n)$
NL	Non-deterministic Turing machine	Space O(log n)
NPSPACE	Non-deterministic Turing machine	Space poly(n)
NEXPSPACE	Non-deterministic Turing machine	Space $2^{\text{poly}(n)}$



A representation of the relation among complexity classes

Wikipedia

The Limits of Science and Logic

Definition

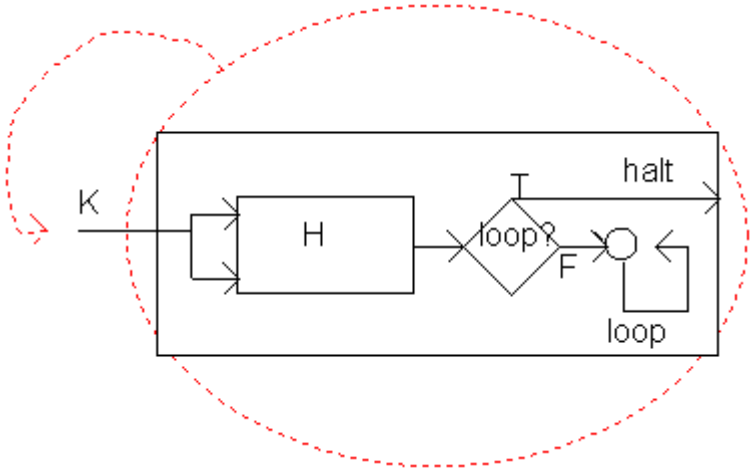
Unsolvable Problem:

A problem for which there exists no algorithm for its solution.

Examples of Unsolvable Problems

- **The Halting Problem**:

  Given a description of an arbitrary computer program, decide whether the program finishes running or continues to run forever.



Examples of Unsolvable Problems: The Halting Problem

The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation, i.e. all programs that can be written in some given programming language that is general enough to be equivalent to a Turing machine. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations on the amount of memory or time required for the program's execution; it can take arbitrarily long, and use arbitrarily much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudocode, the program:

```
while (true) continue
```

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

```
print "Hello, world!"
```

does halt.

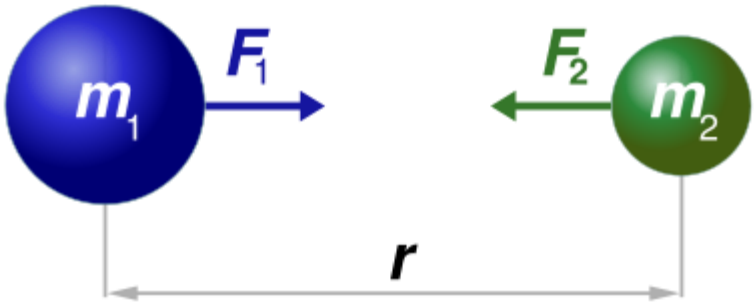
While deciding whether these programs halt is simple, more complex programs prove problematic.

One approach to the problem might be to run the program for some number of steps and check if it halts. But if the program does not halt, it is unknown whether the program will eventually halt or run forever.

Turing proved no algorithm can exist which will always correctly decide whether, for a given arbitrary program and its input, the program halts when run with that input; the essence of Turing's proof is that any such algorithm can be made to contradict itself, and therefore cannot be correct.

Examples of Unsolvable Problems

- **Newton's Universal Law of Gravity**:



$$F_1 = F_2 = G\frac{m_1 \times m_2}{r^2}$$

<u>Two</u> gravitating bodies - SOLVABLE

Examples of Unsolvable Problems

- **The Three-Body Problem**:

  There is no set of equations for the general problem of the gravitational dynamics of three masses.

Examples of Unsolvable Problems

- **The Three-Body Problem**:

  There is no set of equations for the general problem of the gravitational dynamics of three masses - **UNSOLVABLE**.

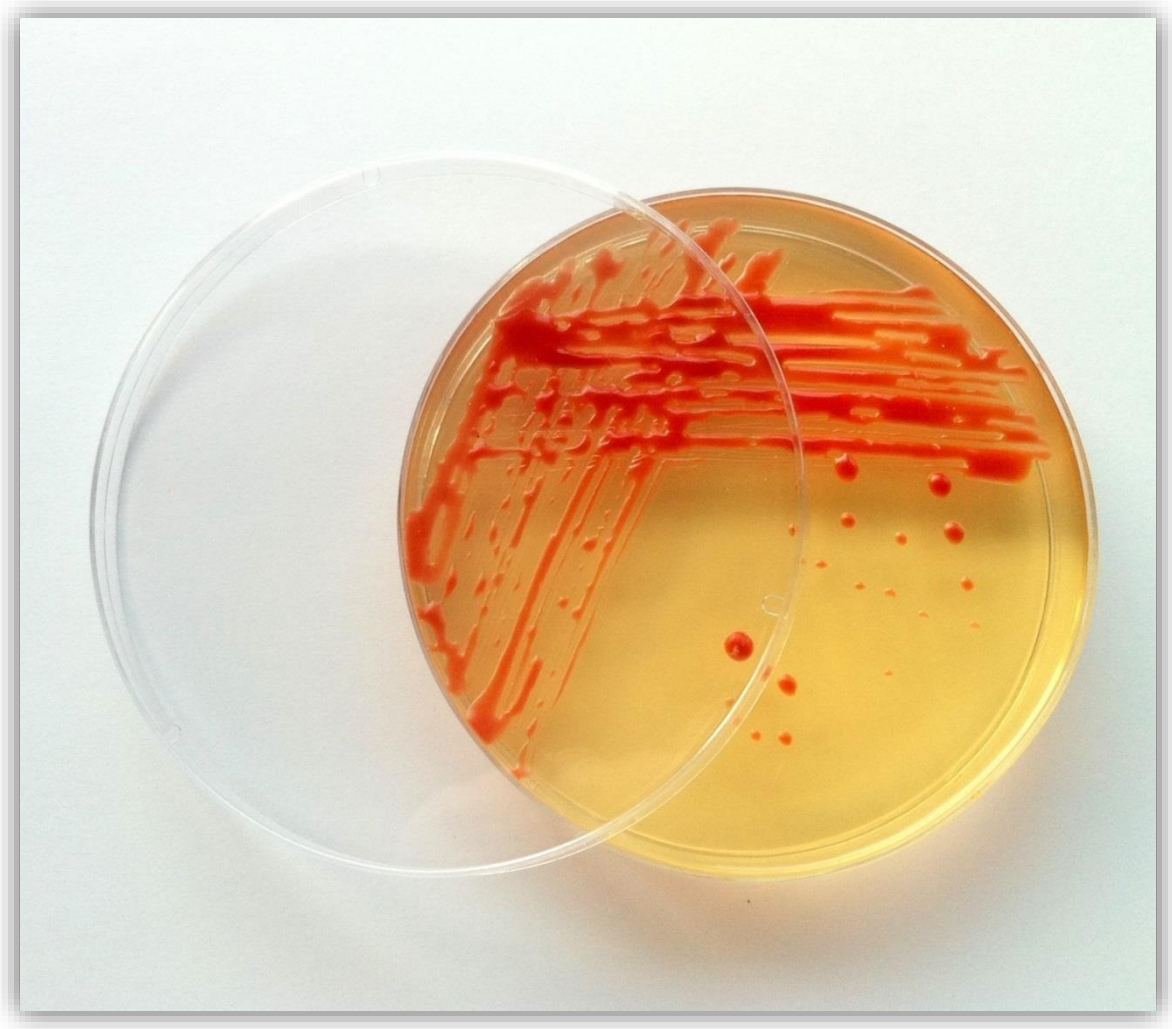
This has to do with, so-called deterministic chaos theory – sensitivity to initial conditions (a.k.a., The Butterfly Effect)

Scientific American

Confronting Science's Logical Limits; October 1996; Scientific American Magazine; by Casti; 4 Page(s)

To anyone infected with the idea that the human mind is unlimited in its capacity to answer questions, a tour of 20th-century mathematics must be rather disturbing. In 1931 Kurt Gödel set forth his *incompleteness theorem*, which established that no system of deductive inference can answer all questions about numbers. A few years later Alan M. Turing proved an equivalent assertion about computer programs, which states that there is no systematic way to determine whether a given program will ever halt when processing a set of data [The Halting Problem]. More recently, Gregory J. Chaitin of IBM has found arithmetic propositions whose truth can never be established by following any deductive rules.

A Final Thought



A Profound Hypothesis

Proposition: Human population will undergo significant social disorientation this century (Malthusian Dynamics - 1798).

A Profound Hypothesis

Proposition: Human population will undergo significant social disorientation this century (Malthusian Dynamics).

Proof (Malthusian):

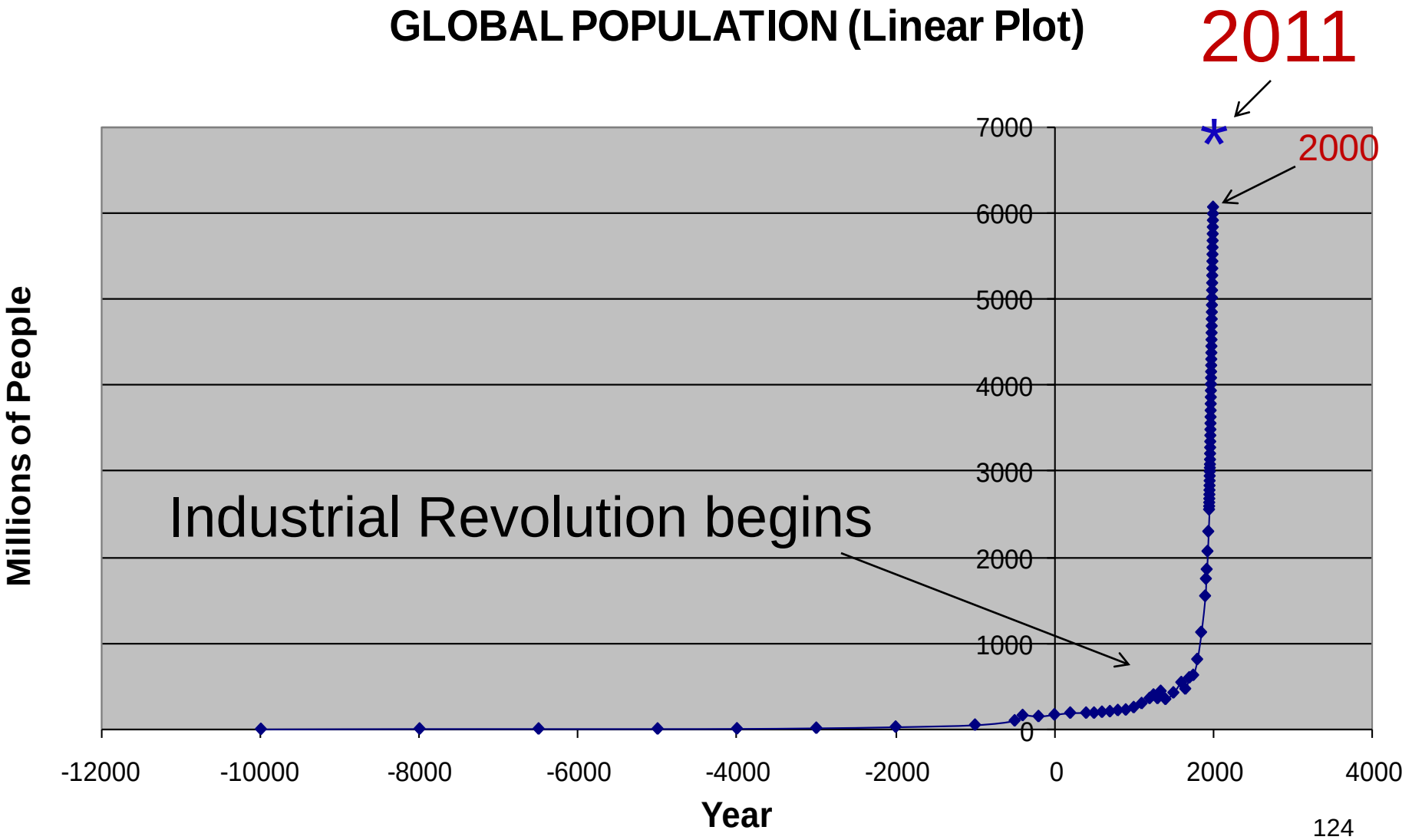
Proposition #1: Animal's reproduce **exponentially**.

Proposition #2: Animal food supply increases **arithmetically**.

Conclusion: There will be have to be a change in P1.

QED

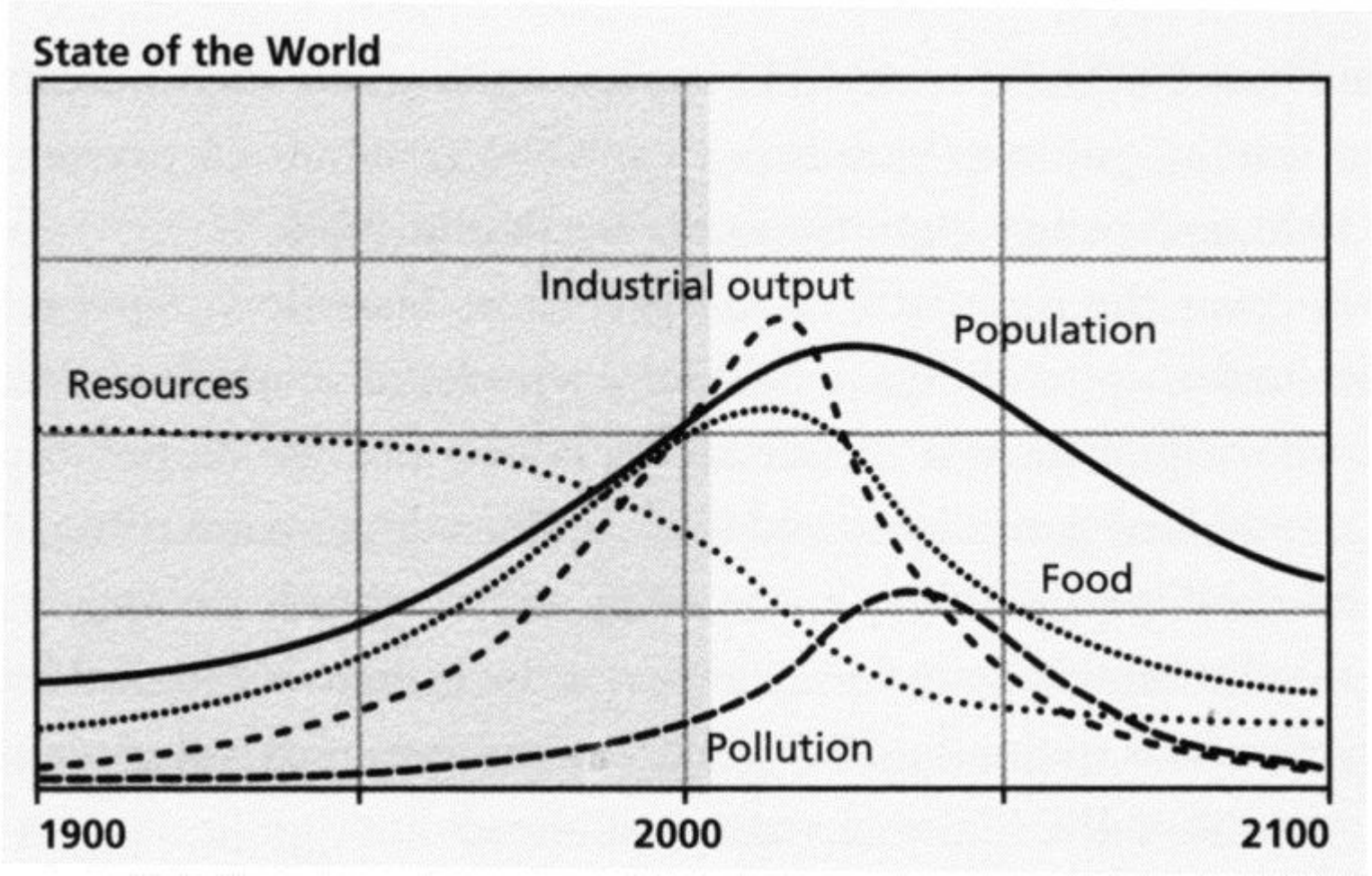
Who, Where, and When?



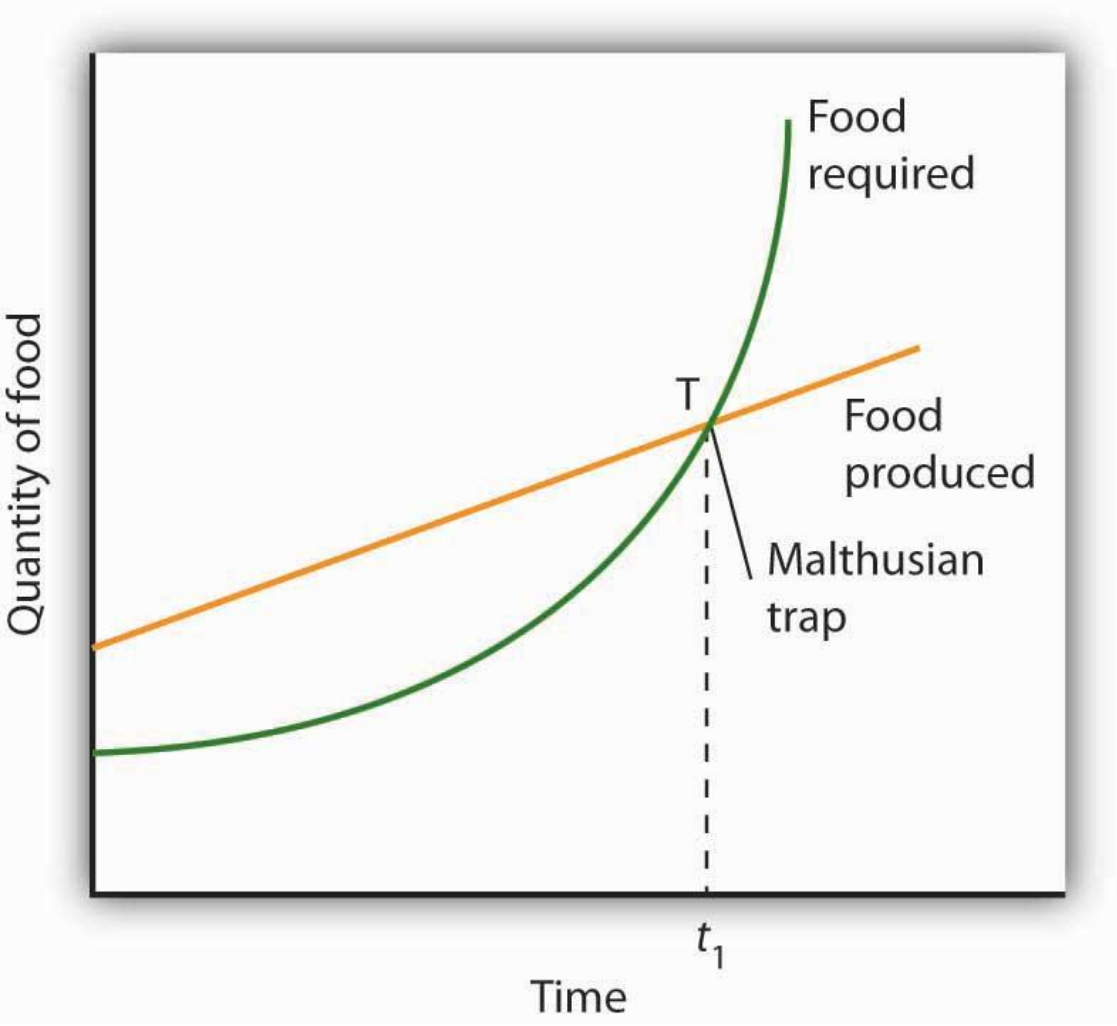
The Malthusian Prediction

http://en.wikipedia.org/wiki/Malthusian_catastrophe

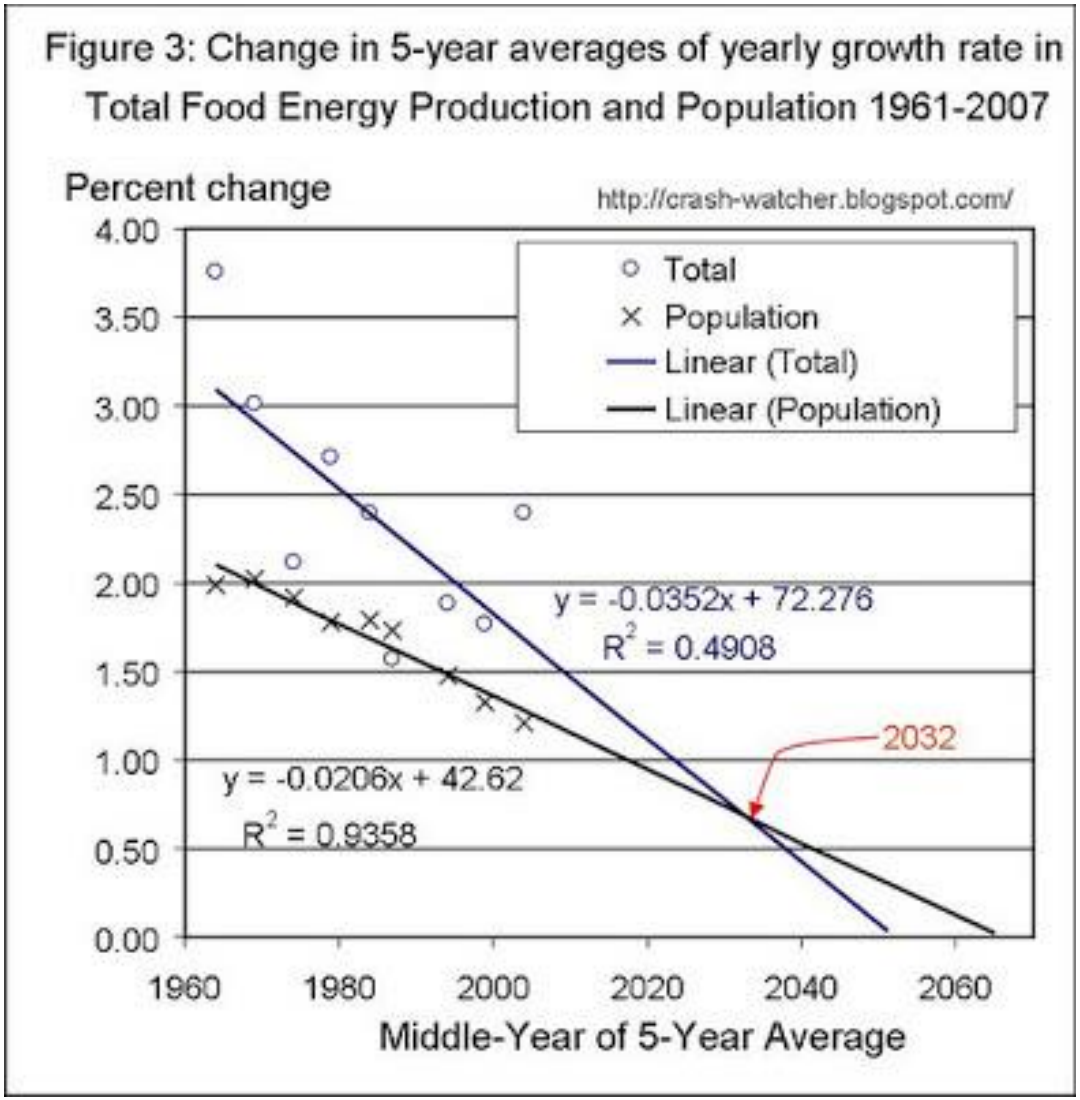
The Club of Rome: Limits to Growth



“Principles of Macroeconomics”



The Year 2030



An Upper Bound on Time

- Assuming that human population is growing exponentially at a doubling time, *d*, of roughly 50 years, how long before we "hit the walls of the petri dish"?
  - Let *L be the* total land mass of Earth: 150 million km^2,
  - Let *D* be the density of "closely packed people": 10^6/ km^2.
  - Then *LD* = $1.5x10^{14}$ humans.
  - Thus: $2^d \cong 10^{14}$ humans.
  - Conclusion: Less than 50 doubling times (2500 years).

CONCLUSION

This is a mathematical proof that human society will necessarily undergo some sort of significant social "reorientation" or worse, and likey during this century.


THE END