Databases and Algorithms

03-Analysis of Algorithms and Asymptotic Notation

Instructor: Rossano Schifanella

@SDS

Algorithms

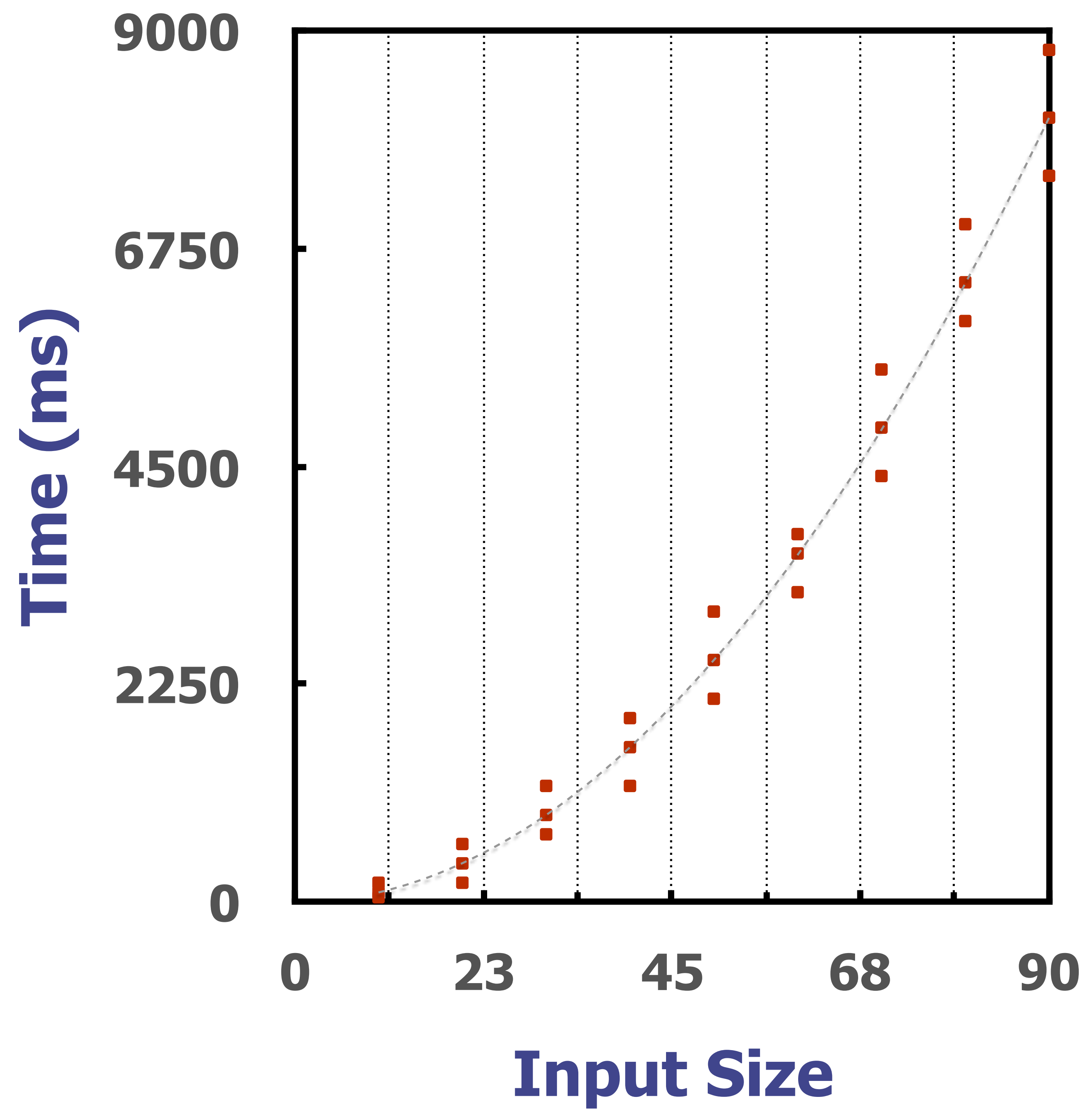
- **What can we analyze about an algorithm?**
    - Make a list

What is an **efficient** algorithm

- Possible efficiency measures
  - Total amount of time on a stopwatch?
  - Low memory usage?
  - Low power consumption?
  - Network usage?
- The analysis of algorithms helps us **quantify** this

Measuring Running Time

- **Experimentally?**
  - Implement algorithm
  - Run algorithm on inputs of different size
  - Measure the running time
  - Plot the results



ARE WE DONE?

Measuring Running Time

- What if you can't implement algorithm?
- Which inputs exactly should you choose?
- Which hardware should you run on?
- Which operating system?
- Which compiler?
- Which compiler flags?
- ...

Measuring Running Time

- We need a measure that is
  - independent of hardware
  - independent of OS
  - independent of compiler
  - ...
- It should depend only on
  - **intrinsic properties of the algorithm**

Knuth’s Observation

- Running time can be determined using
  - Time/cost of each operation
  - Frequency of each operation
- Example:
  - function that sums 100 integers
  - time(sum) = time(read)·100 + time(add)·99
- **Key insight**
  - cost of operations depend on hardware, OS, compiler,...
  - frequency of operations depend on algorithm

What operations? Elementary Operations

- Algorithmic running time is measured in **elementary operations**
  - Math: `+,-,*,/,max,min,log,sin,cos,abs,...`
  - Comparisons: `==,>,<,≤,≥`
  - Variable assignment
  - Variable increment or decrement
  - Array allocation
  - Creating a new object
  - Function calls and value returns
  - Careful: an object's constructor & function calls may have elementary ops too!
- In practice all these operations take different amounts of time
  - **in algorithm analysis we assume each operation takes 1 unit of time**

Towards an **Algorithmic** Running Time

- **Problem #1**
  - running time varies with hardware, OS and so on
  - **solution #1:** focus on number of operations
- **Problem #2**
  - number of operations varies with input size
  - **solution #2:** focus on number of operations for large inputs
- **Problem #3**
  - number of operations varies with input
  - **solution #3:** focus on number of operations on worst-case inputs

Towards an **Algorithmic** Running Time

- Why worst-case inputs?
  - Easier to analyze
  - Gives useful information
    - what if a plane autopilot program runs slower than predicted due to an unexpected input?
- Why large inputs?
  - Easier to analyze
  - We usually care what happens on large data
  - Allows us to ignore odd behaviors that happen on small data

Constant Running Time

```
function first(array):
    // Input: an array
    // Output: the first element
    return array[0]
```

2ops

- How many operations are executed?
  - What if array has **100** elements?
  - What if array has **100,000** elements?
  - What if array has **n** elements?
- **key observation:** running time does not depend on array size

```
function argmax(array)
  // Input: an array
  // Output: the index of the maximum value
  index = 0
  for i in [1, array.length):
    if array[i] > array[index]:
     index = i
  return index
```

Code	Operations
`index = 0`	1op
`for i in [1, array.length):`	1op per loop
`if array[i] > array[index]:`	3ops per loop
`index = i`	1op per loop (sometimes)
`return index`	1op

Linear Running Time

```
function argmax(array)
  // Input: an array
  // Output: the index of the max value
  index = 0
  for i in [1, array.length):
    if array[i] > array[index]:
     index = i
  return index
```

1op

1op per loop

3ops per loop

1op per loop
(sometimes)

1op

- How many operations are executed?
  - What if array has `10` elements?
  - What if array has `100,000` elements?
- **key observation:** running time depends on array size
  - `5n+2` operations where `n=size(array)`

```
function possible_products(array):
    // Input: an array
    // Output: a list of all possible products
    //         between any two elements in the list
    products = []
    for i in [0, array.length):
        for j in [0, array.length):
            products.append(array[i] * array[j])
    return products
```

1op

1op per loop

1op per loop
per loop

4ops per loop
per loop

1op

Quadratic Running Time

```
function possible_products(array):
    // Input: an array
    // Output: a list of all possible products
    //          between any two elements in the list
    products = []                          <- 1op
    for i in [0, array.length):            <- 1op per loop
        for j in [0, array.length):        <- 1op per loop per loop
            products.append(array[i] * array[j])  <- 4ops per loop per loop
    return products                        <- 1op
```

- **key observation:** running time depends on the **square** of array size
  - $5n^2+n+2$ operations where `n=size(array)`

Growth of Functions

- We are usually interesting in the **order of growth** of the running time of an algorithm, not in the exact running time. This is also referred to as the **asymptotic running time**.
- We need to develop a way to talk about rate of growth of functions so that we can compare algorithms.
- **Asymptotic notation** gives us a method for classifying functions according to their rate of growth.

Asymptotic notations

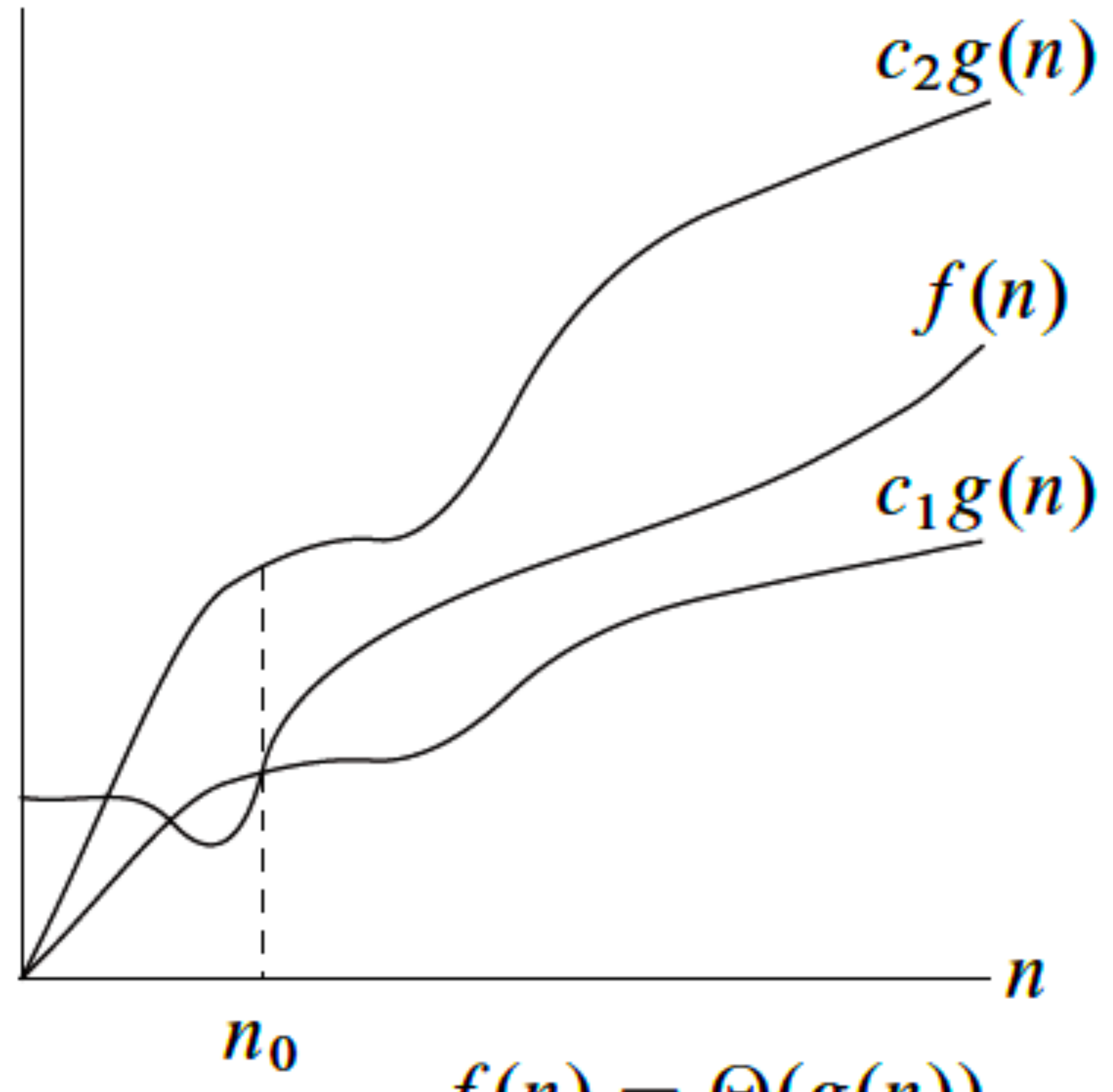
- Θ-notation (theta)
- O-notation (big-oh)
- Ω-notation (big-omega)
- o-notation (little-oh)
- ω-notation (little-omega)

Θ-notation (theta)

f(n) = Θ(g(n) : THE EQUAL SIGN = MEANS IN REALITY SET MEMBERSHIP ∈

$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ $\}$

g(n) IS AN ASYMPTOTIC TIGHT BOUND FOR f(n)



$f(n) = \Theta(g(n))$

Example

$n^2 + 5n + 7 = \Theta(n^2)$

When $n \geq 1$,

$n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$

When $n \geq 0$,

$n^2 \leq n^2 + 5n + 7$

Thus, when $n \geq 1$

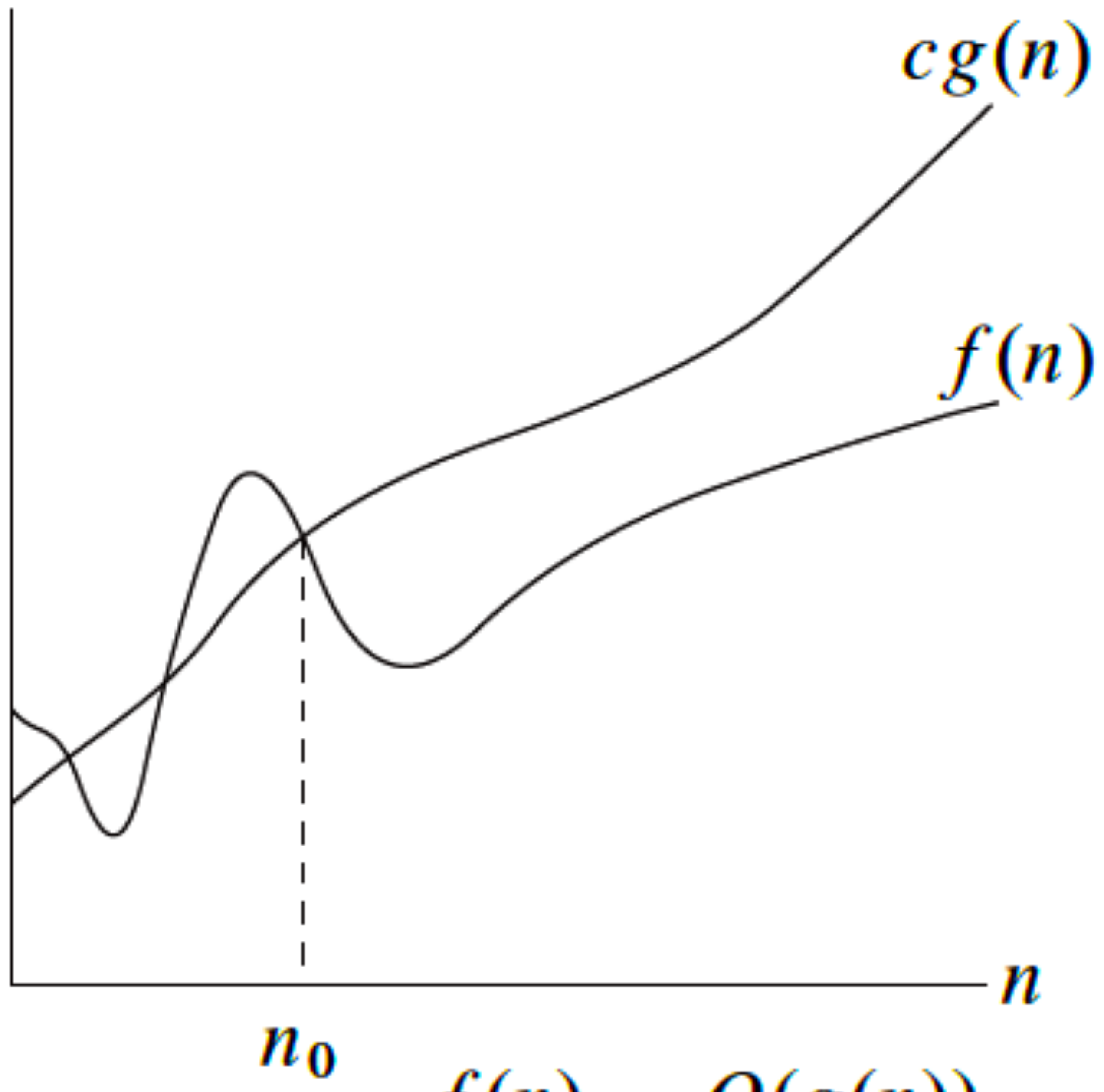
$1n^2 \leq n^2 + 5n + 7 \leq 13n^2$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Θ, with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$.)

O-notation (big-oh)

$O(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

g(n) IS AN ASYMPTOTIC UPPER BOUND FOR f(n)



$f(n) = O(g(n))$

EXAMPLE

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

Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$

Notice that if $n \geq 1$, $n \leq n^3$ is clear.

Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.

Side Note: In general, if $a \leq b$, then $n^a \leq n^b$ whenever $n \geq 1$. This fact is used often in these types of proofs.

Therefore:

$n^2 + n \leq n^3 + n^3 = 2n^3$

We have just shown that

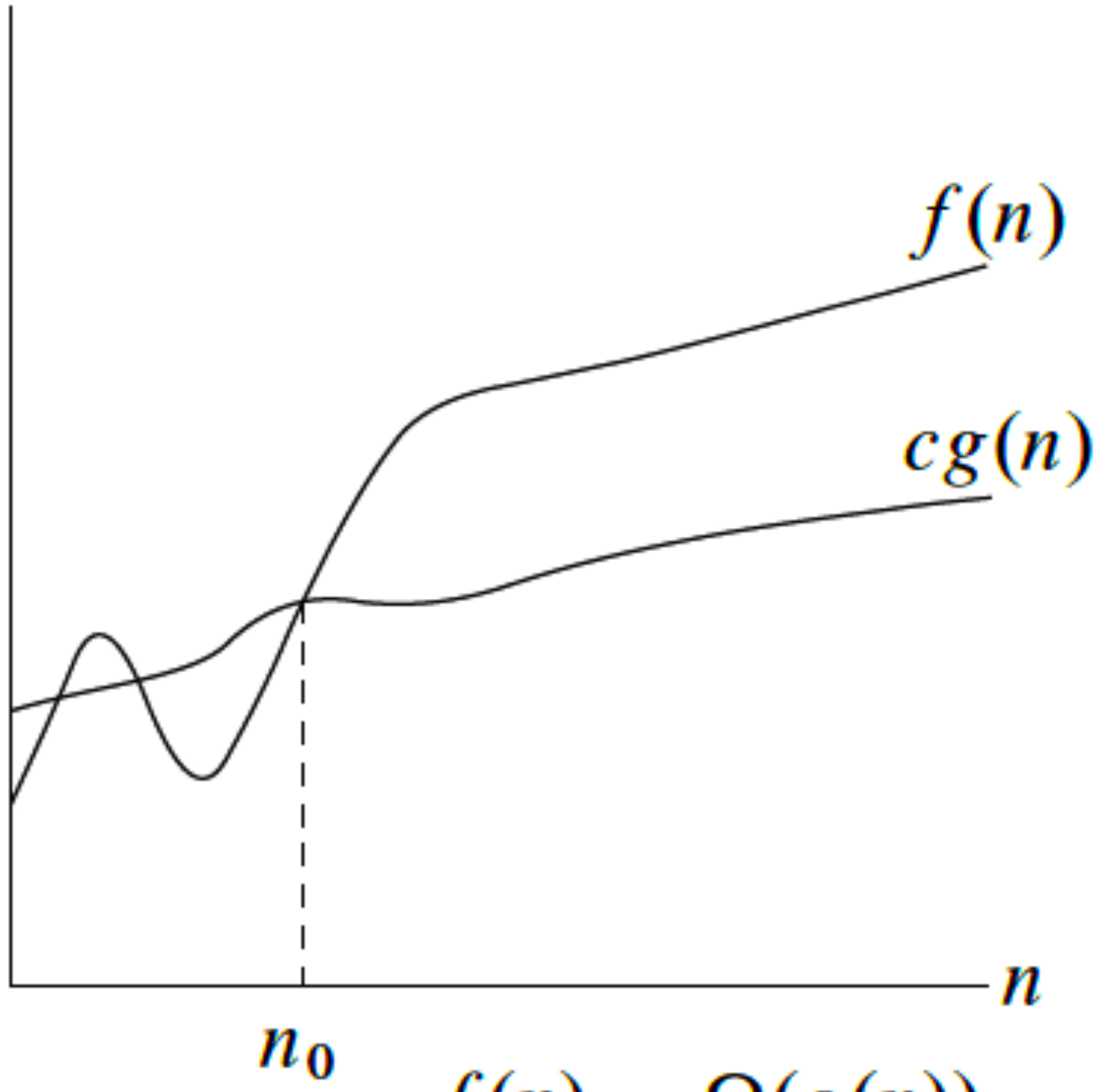
$n^2 + n \leq 2n^3$ for all $n \geq 1$

Thus, we have shown that $n^2 + n = O(n^3)$ (by definition of Big-Oh, with $n_0 = 1$, and $c = 2$.)

Ω-notation (big-omega)

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$

g(n) IS AN ASYMPTOTIC LOWER BOUND FOR f(n)



$f(n) = \Omega(g(n))$

EXAMPLE

$n^3 + 4n^2 = \Omega(n^2)$

Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$

It is not too hard to see that if $n \geq 0$,

$n^3 \leq n^3 + 4n^2$

We have already seen that if $n \geq 1$,

$n^2 \leq n^3$

Thus when $n \geq 1$,

$n^2 \leq n^3 \leq n^3 + 4n^2$

Therefore,

$1n^2 \leq n^3 + 4n^2$ for all $n \geq 1$

Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big-Omega, with $n_0 = 1$, and $c = 1$.)

Arithmetic of Θ, O, Ω

- **Transitivity:**

  $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

  $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

  $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

- **Scaling:**

  if $f(n) = O(g(n))$ then for any $k > 0$, $f(n) = O(kg(n))$

- **Sums:**

  if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $(f_1 + f_2)(n) = O(\max(g_1(n), g_2(n)))$

Arithmetic of Θ, O, Ω

- <u>**Reflexivity:**</u>

  $f(n) = \Theta(f(n))$

  $f(n) = O(f(n))$

  $f(n) = \Omega(f(n))$

- <u>**Simmetry:**</u>

  $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

Strategies for O

- Sometimes the easiest way to prove that $f(n) = O(g(n))$ is to take c to be the sum of the positive coefficients of $f(n)$.
- We can usually ignore the negative coefficients. Why?
- **Example:** To prove $5n^2 + 3n + 20 = O(n^2)$, we pick $c = 5 + 3 + 20 = 28$. Then if $n \geq n_0 = 1$,

  $$5\,n^2 + 3\,n + 20 \leq 5\,n^2 + 3\,n^2 + 20\,n^2 = 28\,n^2,$$

  thus $5n^2 + 3n + 20 = O(n^2)$.
- This is not always so easy. How would you show that $(\sqrt{2})^{\log n} + \log^2 n + n^4$ is $O(2^n)$? Or that $n^2 = O(n^2 - 13n + 23)$? After we have talked about the relative rates of growth of several functions, this will be easier.
- In general, we simply (or, in some cases, with much effort) find values c and n_0 that work. This gets easier with practice.

Strategies for Θ, Ω

- Proving that a $f(n) = \Omega(g(n))$ often requires more thought.
  - Quite often, we have to pick $c < 1$.
  - A good strategy is to pick a value of c which you think will work, and determine which value of n_0 is needed.
  - Being able to do a little algebra helps.
  - We can sometimes simplify by ignoring terms if $f(n)$ with the positive coefficients. Why?
- The following theorem shows us that proving $f(n) = \Theta(g(n))$ is nothing new:
  - **Theorem:** $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - Thus, we just apply the previous two strategies.

Summary

- It is important to remember that a Big-O bound is only an <u>**upper**</u> bound. So an algorithm that is $O(n^2)$ might not ever take that much time. It may actually run in $O(n)$ time.
- Conversely, an Ω bound is only a <u>**lower**</u> bound. So an algorithm that is $\Omega(n \log n)$ might actually be $\Theta(2^n)$.
- Unlike the other bounds, a Θ-bound is <u>**precise**</u>. So, if an algorithm is $\Theta(n^2)$, it runs in quadratic time.

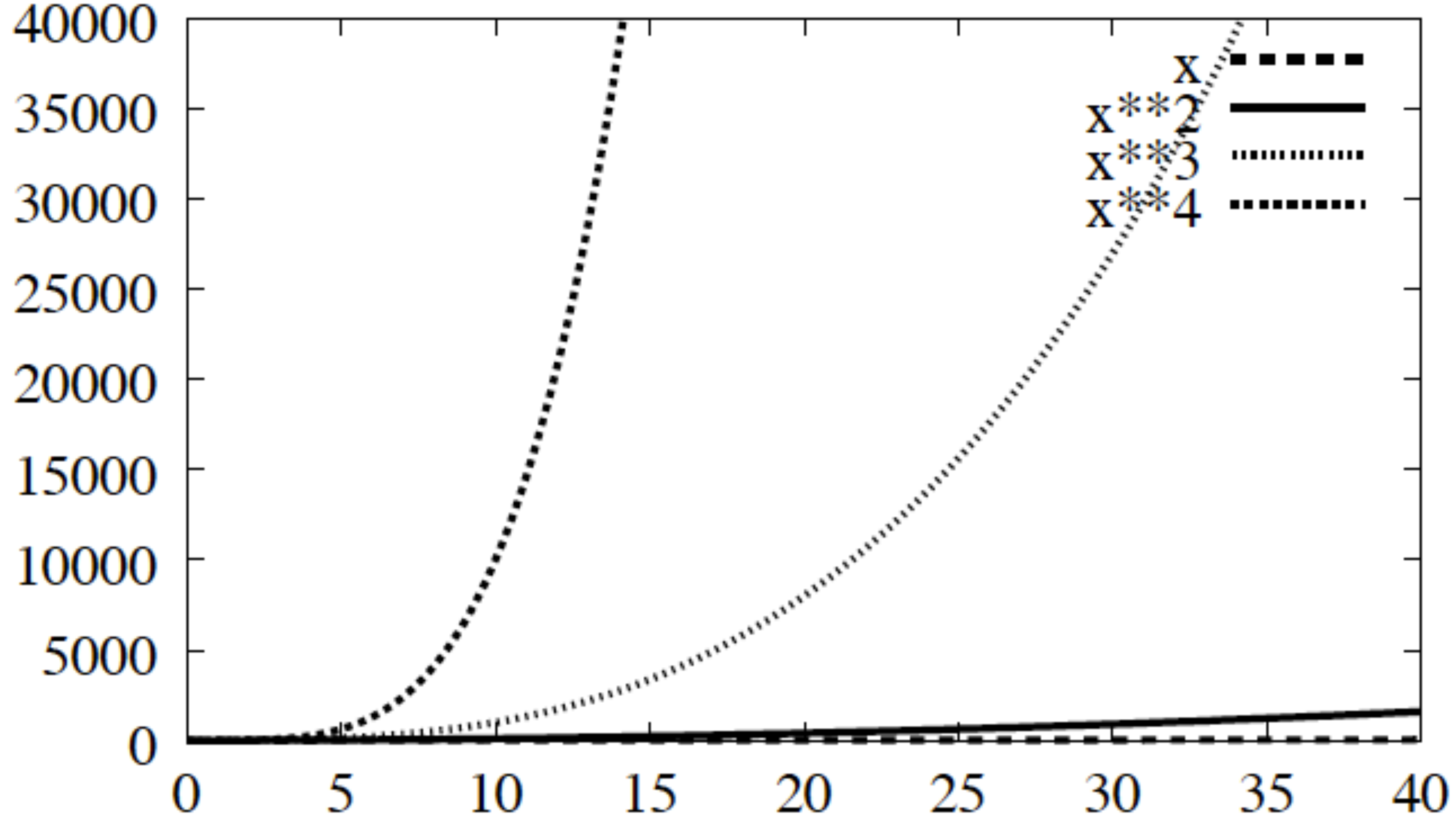
Common rates of growth

- Let n be the size of input to an algorithm, and k some constant. The following are common rates of growth:
  - **Constant**: $\Theta(k)$, for example $\Theta(1)$
  - **Linear**: $\Theta(n)$
  - **Logarithmic**: $\Theta(\log_k n)$
  - **n log n**: $\Theta(n \log_k n)$
  - **Quadratic**: $\Theta(n^2)$
  - **Polynomial**: $\Theta(n^k)$
  - **Exponential**: $\Theta(k^n)$

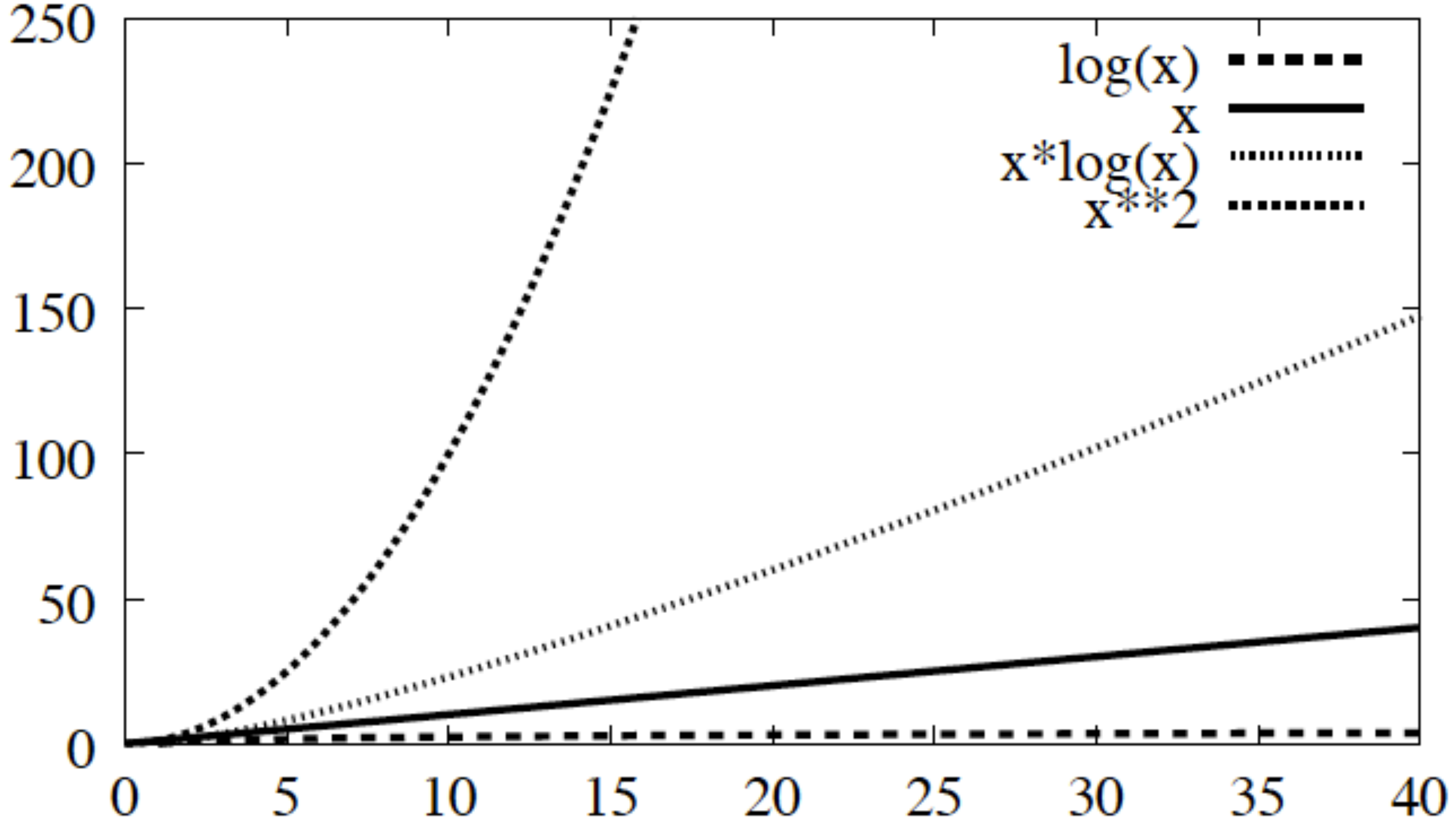
THE FOLLOWING INEQUALITIES HOLD ASYMPTOTICALLY:

$c < \log n < \log^2 n < \sqrt{n} < n < n \log n < n^{(1.1)} < n^2 < n^3 < n^4 < 2^n$

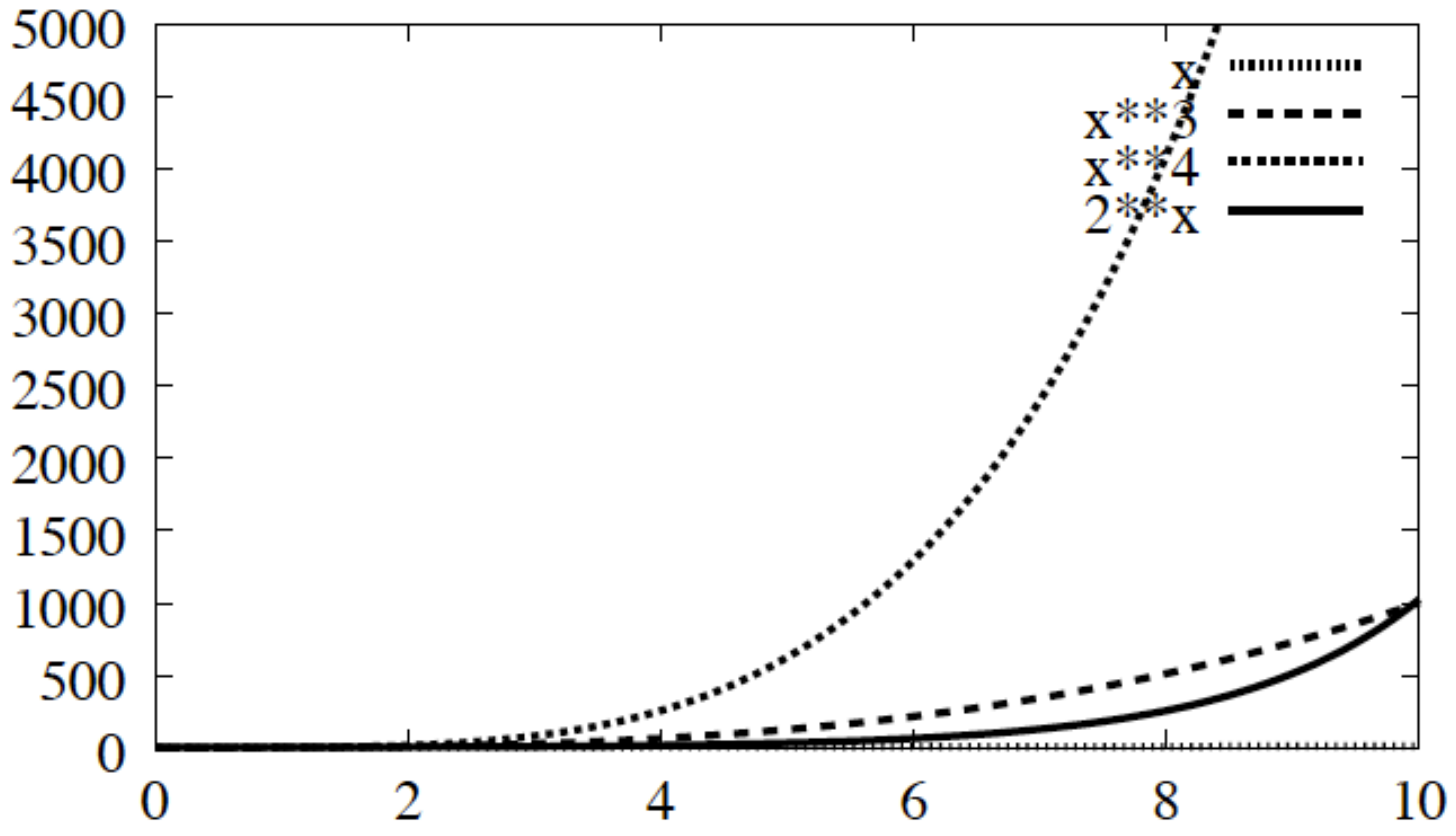
Polynomial functions



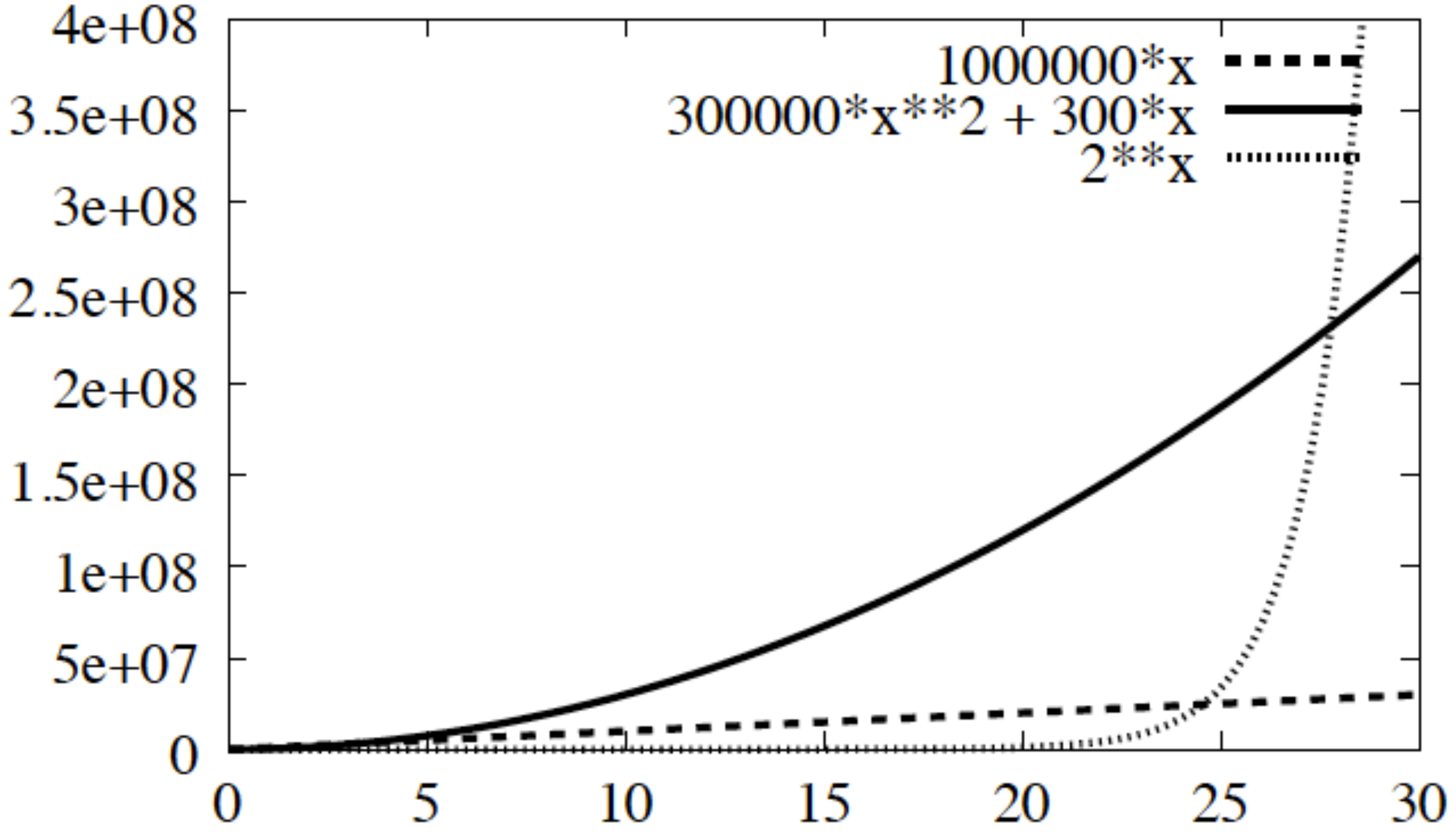
Slow growing functions



Fast growing functions



Why Constants and Non-Leading Terms Don’t Matter



Questions?

@rschifan

schifane@di.unito.it

http://www.di.unito.it/~schifane