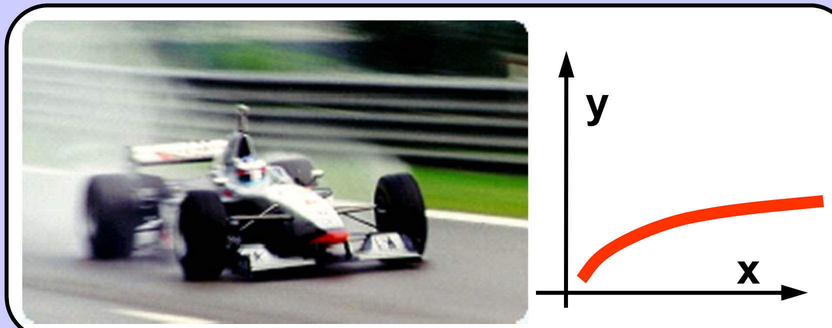
The complexity of different algorithms varies

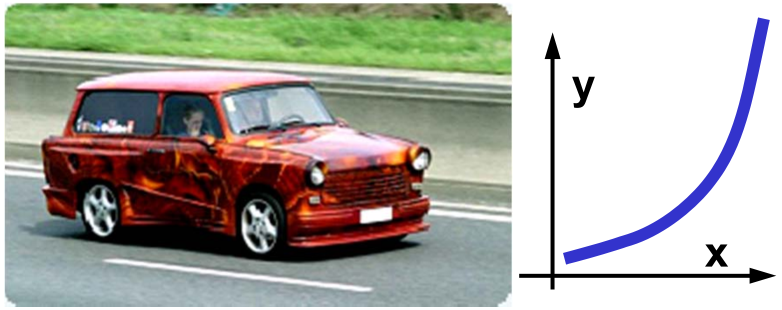
The speed…

One algorithm (program, method…)
is **faster** than another one.

What do we mean by this statement??

Asymptotic complexity





Each algorithm can be unambiguously assigned

growing function

named

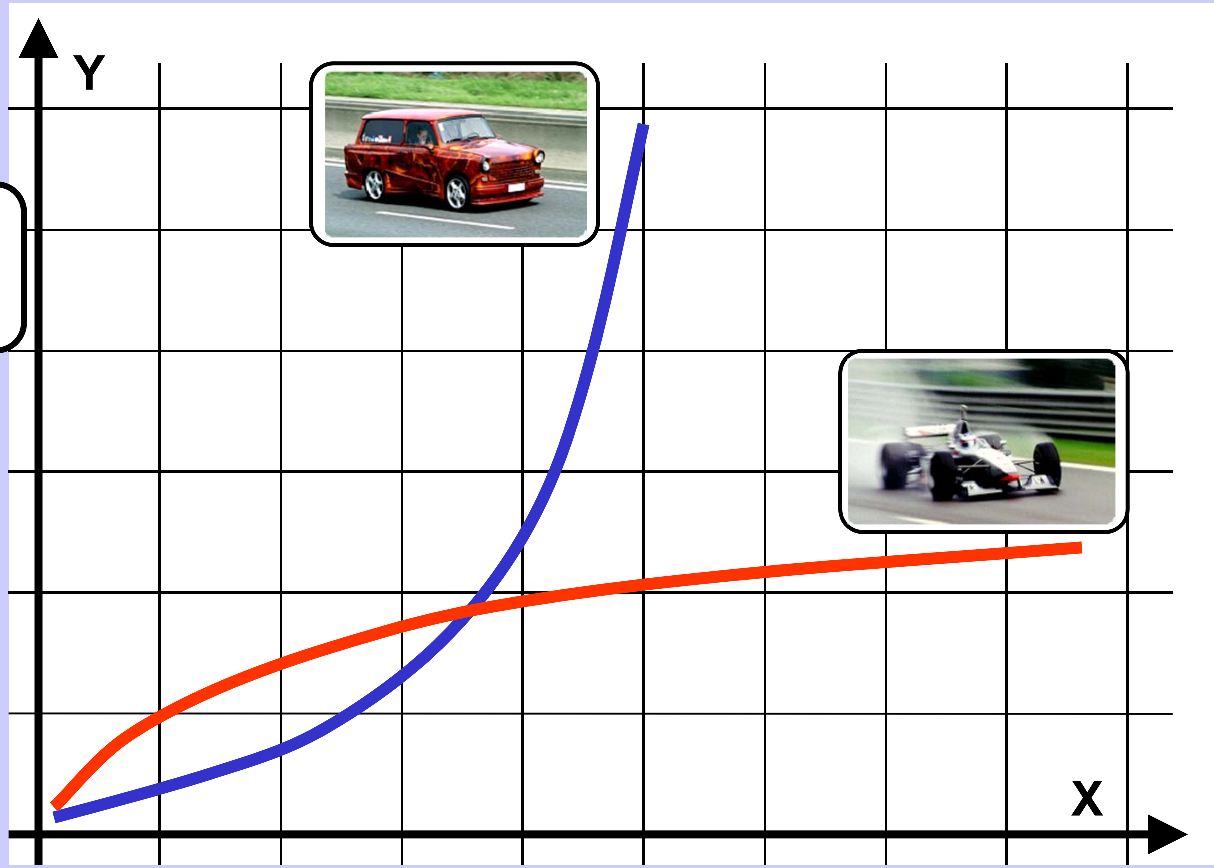
asymptotic complexity

which characterizes the number of algorithm operations with respect to the growing size of input data.

The slower this function grows the faster the algorithm.

Asymptotic complexity

Y ~ system load
(computing time)



x ~ our demands
(input data size)

Examples

Find min and max value in an array — STANDARD

min	max	a								
3	3	**3**	2	7	10	0	5	–10	4	6

min	max	a								
3	3	3	**2**	7	10	0	5	–10	4	6

```
if a[i] < min: min = a[i]
if a[i] > max: max = a[i]
```

min	max	a								
2	3	3	**2**	7	10	0	5	–10	4	6

Examples

Find min and max value in an array — STANDARD

min max a

min	max
2	7

3	2	7	10	0	5	–10	4	6

etc...

done

min	max
–10	10

3	2	7	10	0	5	–10	4	6

code

```
min = a[0]; max = a[0]
for i in range( 1, len(a) ):
   if a[i] < min: min = a[i]
   if a[i] > max: max = a[i]
```

Examples

Find min and max value in an array — FASTER!

min	max
3	3

a

3	2	7	10	0	5	–10	4	6

min	max
3	3

a

3	2	7	10	0	5	–10	4	6

```
if a[i] < a[i+1]:
   if   a[i] < min: min = a[i]
   if a[i+1] > max: max = a[i+1]
```

min	max
2	7

a

3	2	7	10	0	5	–10	4	6

Examples

Find min and max value in an array — FASTER!

min	max
2	7

a

3	2	7	10	0	5	–10	4	6

```
if a[i] < a[i+1]:
   if    a[i] < min: min = a[i]
   if a[i+1] > max: max = a[i+1]
else:
   if    a[i] > max: max = a[i]
   if a[i+1] < min: min = a[i+1]
```

min	max
0	10

a

3	2	7	10	0	5	–10	4	6

Examples

Find min and max value in an array — FASTER!

done

min	max
–10	10

a

3	2	7	10	0	5	–10	4	6

code

step=2

```
min = a[0]; max = a[0]
for i in range(1, len(a)-1, 2):
   if a[i] < a[i+1]:
      if    a[i] < min: min = a[i]
      if a[i+1] > max: max = a[i+1]
   else:
      if    a[i] > max: max = a[i]
      if a[i+1] < min: min = a[i+1]
```

Computing the complexity

Elementary operation

- arithmetic operation
- comparison of two numbers
- number move in the memory

Complexity A

a total number of elementrary operations

simplification

Complexity B

a total number of elementary operations on data

Computing the complexity

Complexity B a total number of elementary operations on data

another simplification

Complexity C a total number of number (or character) comparisons on the data

The most common way of computing the complexity

Computing the complexity

Find min and max value in an array — STANDARD

Complexity A

All operations

len(a) = N

```
      1               1
min = a[0]; max = a[0]
       1            N–1
for i in range(1, len(a)):
          N–1         0...N–1
   if a[i] < min: min = a[i]
          N–1         0...N–1
   if a[i] > max: max = a[i]
```

case

best	1 + 1 + 1 + N–1 + N–1 + 0 + N–1 + 0 = 3N
worst	1 + 1 + 1 + N–1 + N–1 + N–1 + N–1 + N–1 = 5N–2

Computing the complexity

Find min and max value in an array — STANDARD

B

Complexity

operations on data

len(a) = N

```
min = a[0]; max = a[0]

for  i in range(1, len(a)):

   if a[i] < min: min = a[i]

   if a[i] > max: max = a[i]
```

Operation counts: min = a[0]: 1; max = a[0]: 1; a[i] < min: N–1; min = a[i]: 0...N–1; a[i] > max: N–1; max = a[i]: 0...N–1

case

best	1 + 1 + N–1 +0 + N–1 + 0 = 2N
worst	1 + 1 + N–1 + N–1 + N–1 + N–1 = 4N–2

Computing the complexity

Find min and max value in an array — FASTER!

Complexity C

only tests on data

len(a) = N

```
min = a[0]; max = a[0]

for i in range(1, len(a)):
   if a[i] < min: min = a[i]
   if a[i] > max: max = a[i]
```

N–1

N–1

always N–1 + N–1 = 2N–2 tests

Computing the compl

Find min and max value in an array — FASTER!

Complexity C

only tests on data

len(a) = N

a, N

3	2	7	10	0	5	–10	4	6

one pair — 3 tests

(N–1)/2 pairs

always $3(N-1)/2 = \underline{(3N-3)/2}$ tests

Computing the complexity

Array size **N**	No. of tests STANDARD **2 (N – 1)**	No. of tests FASTER **(3N – 3)/2**	Ratio STD/FASTER
11	20	15	1.33
21	40	30	1.33
51	100	75	1.33
101	200	150	1.33
201	400	300	1.33
501	1 000	750	1.33
1 001	2 000	1 500	1.33
2 001	4 000	3 000	1.33
5 001	10 000	7 500	1.33
1 000 001	2 000 000	1 500 000	1.33

Tab. 1

Examples

Data

array a:

1	–1	0	–2	5	1	0

array b:

4	2	4	3	4	2	7

Problem

How many elements of array b are equal to the sum of all elements of array a?

Solution

array a:

1	–1	0	–2	5	1	0

sum = 4

array b:

4	2	4	3	4	2	7

result = 3

Examples

built-in function

```
sum(a) #returns sum
```

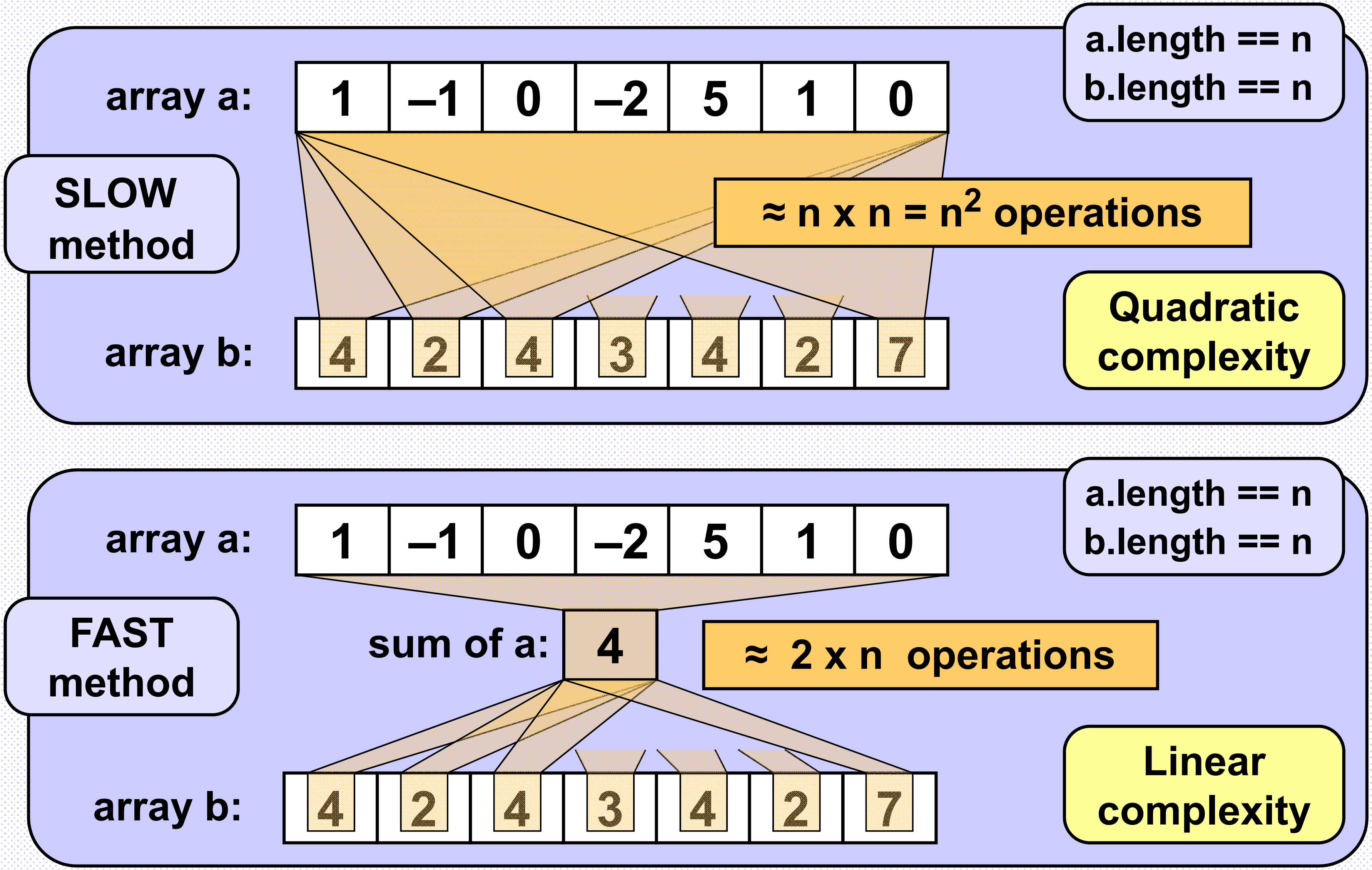
SLOW method

```
count = 0
for i in range(len(b)):
   if b[i]== sum(a): count += 1
return count
```

FAST method

```
count = 0
sumOf_a = sum(a)
for i in range(len(b)):
   if b[i]== sumOf_a : count += 1
return count
```

Computing the complexity



Computing the complexity

Array size N	SLOW method operations N^2	FAST method operations 2N	Ratio SLOW/FAST
11	121	22	5.5
21	441	42	10 .5
51	2 601	102	25 .5
101	10 201	202	50 .5
201	40 401	402	100 .5
501	251 001	1 002	250 .5
1 001	1 002 001	2 002	500 .5
2 001	4 004 001	4 002	1 000 .5
5 001	25 010 001	10 002	2 500 .5
1 000 001	1 000 002 000 001	2 000 002	500 000 .5

Tab. 2

Computing the complexity

Array Size N	Speed ratios solutions of task 1	Speed ratios solutions of task 2
11	1.33	5.5
21	1.33	10 .5
51	1.33	25 .5
101	1.33	50 .5
201	1.33	100 .5
501	1.33	250 .5
1 001	1.33	500 .5
2 001	1.33	1 000 .5
5 001	1.33	2 500 .5
1 000 001	1.33	500 000 .5

Tab. 3

Examples

Search in a sorted array — linear, SLOW

array

sorted array: →

size = N

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

Find 993 !

tests: N

!

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

Find 363 !

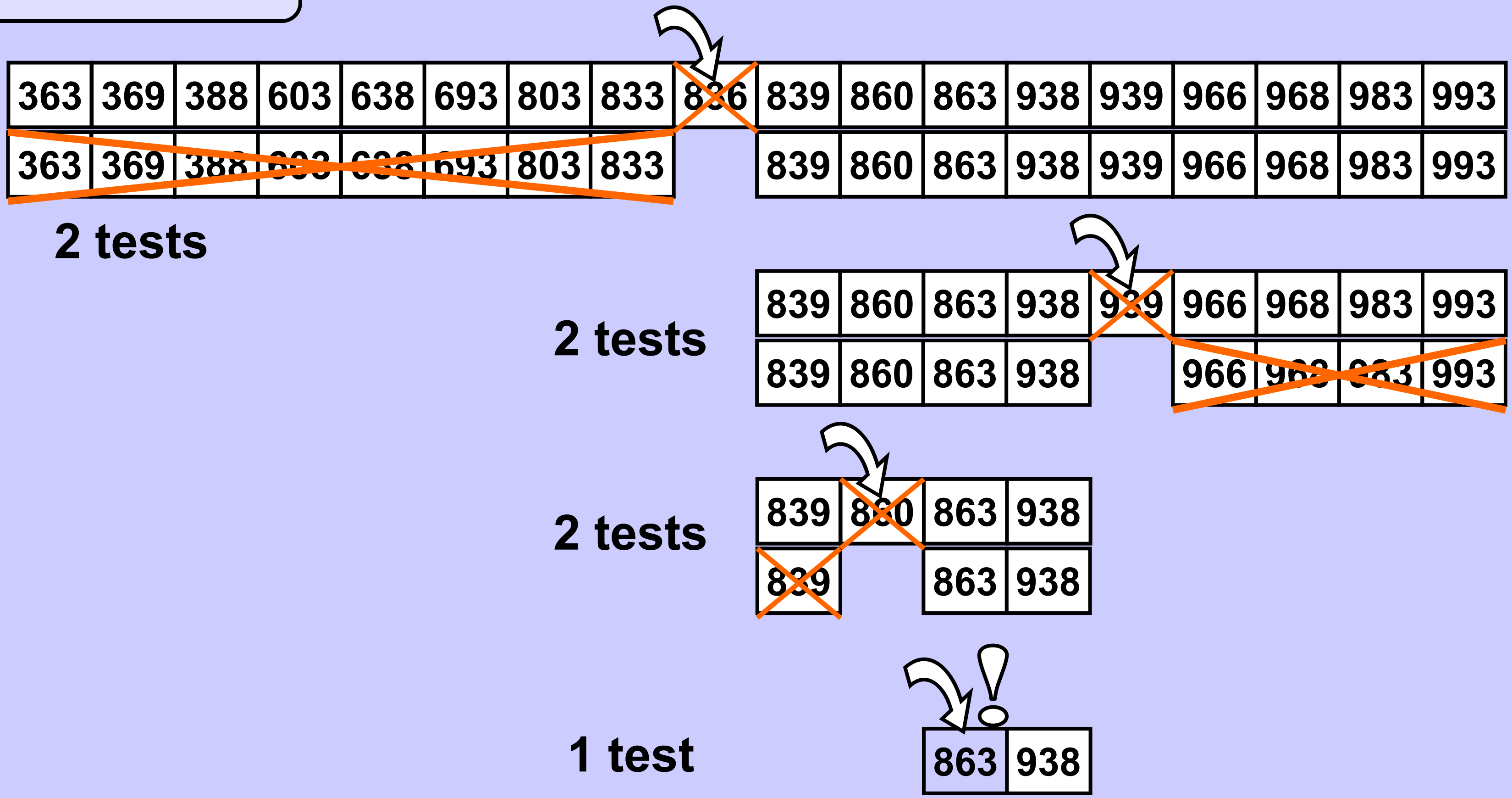
! **tests: 1**

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

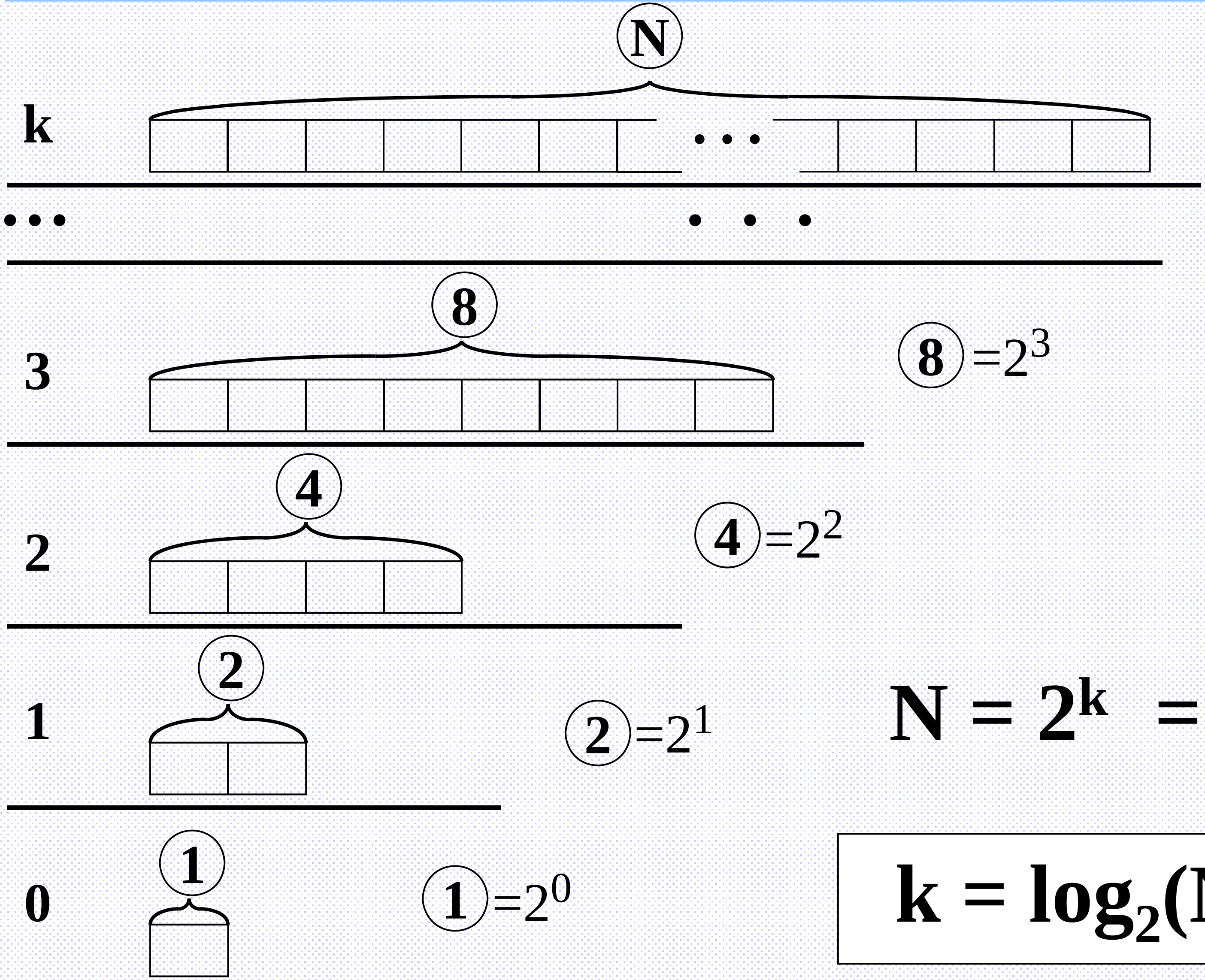
Examples

Search in a sorted array — binary, FAST

Fast 863 !



Exponent, logarithm and interval halving



$N = 2^k$

$8 = 2^3$

$4 = 2^2$

$2 = 2^1$

$1 = 2^0$

$N = 2^k$ =>

$$\boxed{k = \log_2(N)}$$

Computing the complexity

Array size	Number of tests: linear search — case best	worst	average	binary search worst case	ratio
5	1	5	3	5	0.6
10	1	10	5.5	7	0.79
20	1	20	10.5	9	1.17
50	1	50	25.5	11	2.32
100	1	100	50.5	13	3.88
200	1	200	100.5	15	6.70
500	1	500	250.5	17	14.74
1 000	1	1000	500.5	19	26.34
2 000	1	2000	1000.5	21	47.64
5 000	1	5000	2500.5	25	100.02
1 000 000	1	1 000 000	500 000.5	59	8 474.58

Tab. 4

Computing the complexity

The computation time for various time complexities assuming that 1 operation takes 1 μs (10^{-6} sec)

complexity	Size of data					
	10	20	40	60	500	1000
$log_2 n$	3,3 μs	4,3 μs	5 μs	5,8 μs	9 μs	10 μs
n	10 μs	*20* μs	40 μs	60 μs	0,5 ms	1 ms
$n\,log_2 n$	33 μs	86 μs	0,2 ms	0,35 ms	4,5 ms	10 ms
n^2	0,1 ms	0,4 ms	1,6 ms	3,6 ms	0,25 s	1 s
n^3	1 ms	8 ms	64 ms	0,2 s	125 s	17 min
n^4	10 ms	160 ms	2,56 s	13 s	17 h	11,6 days
2^n	1 ms	1 s	12,7 days	36000 yrs	10^{137} yrs	10^{287} yrs
$n!$	3,6 s	77000 yrs	10^{34} yrs	10^{68} yrs	10^{1110} yrs	10^{2554} yrs

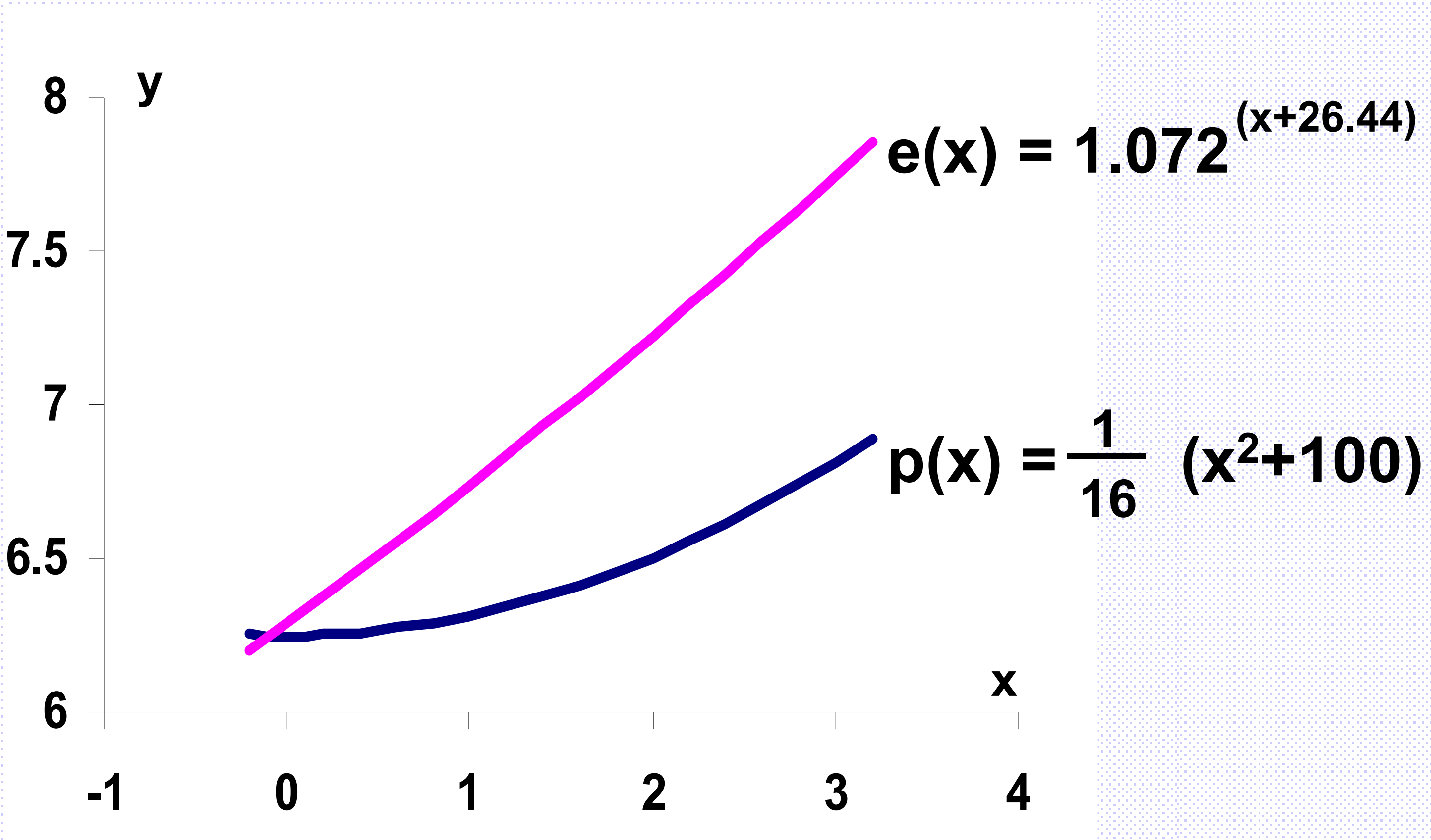
Tab. 5

Functions' order of growth

Ω O Θ

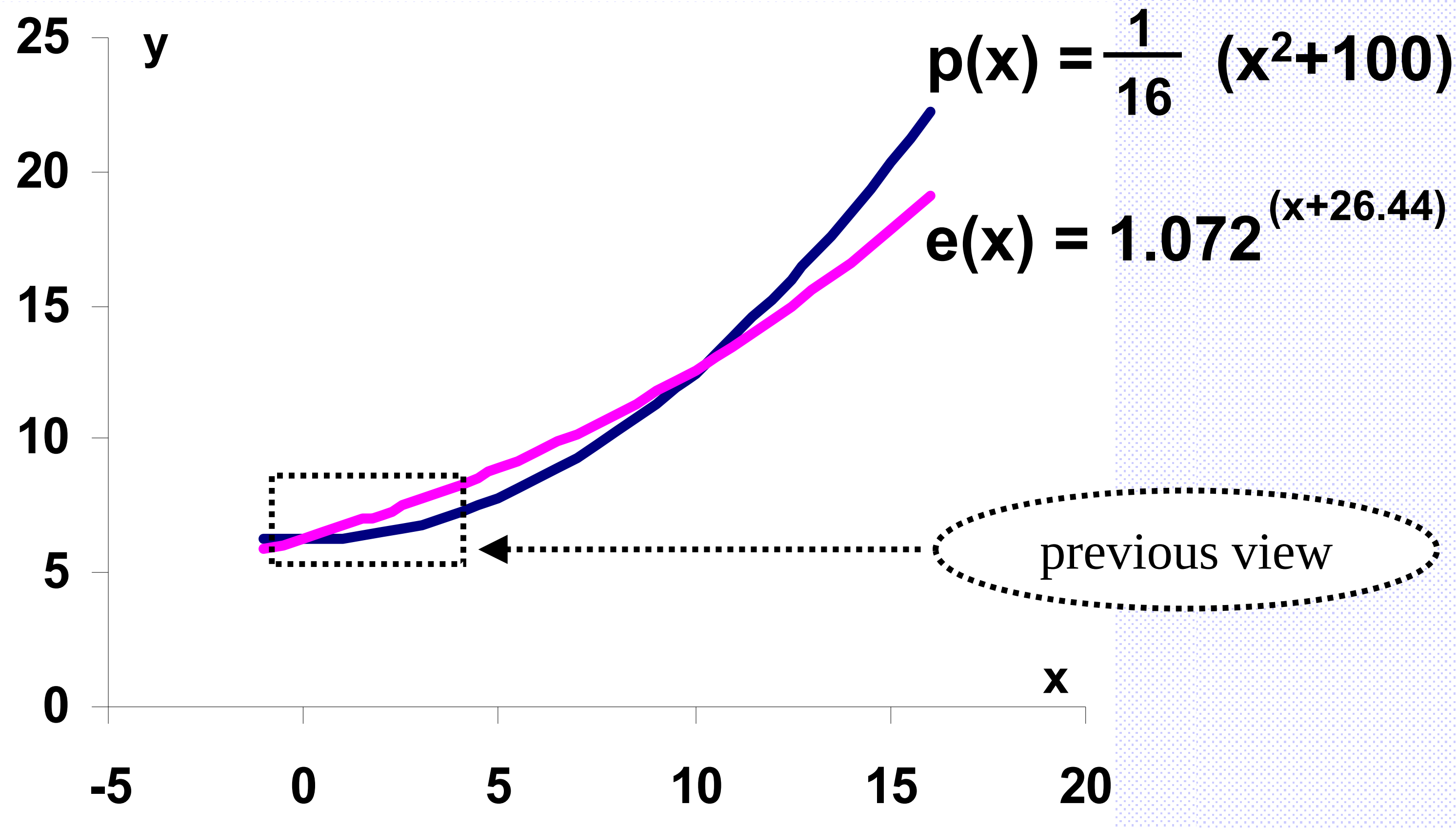
Functions' order of growth

Functions' order of growth



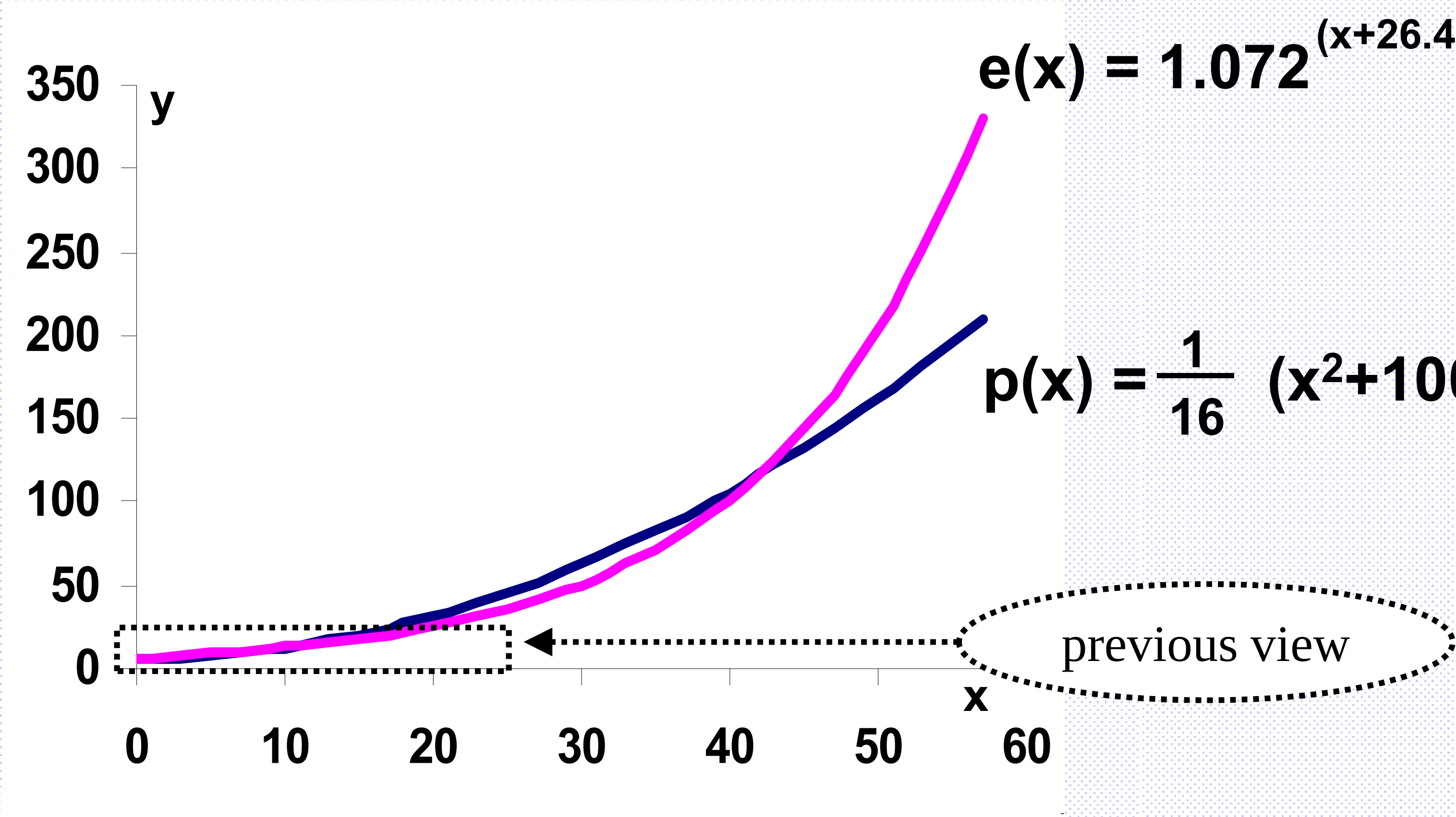
Functions' order of growth

Zoom out! :



Functions' order of growth

Zoom out! :

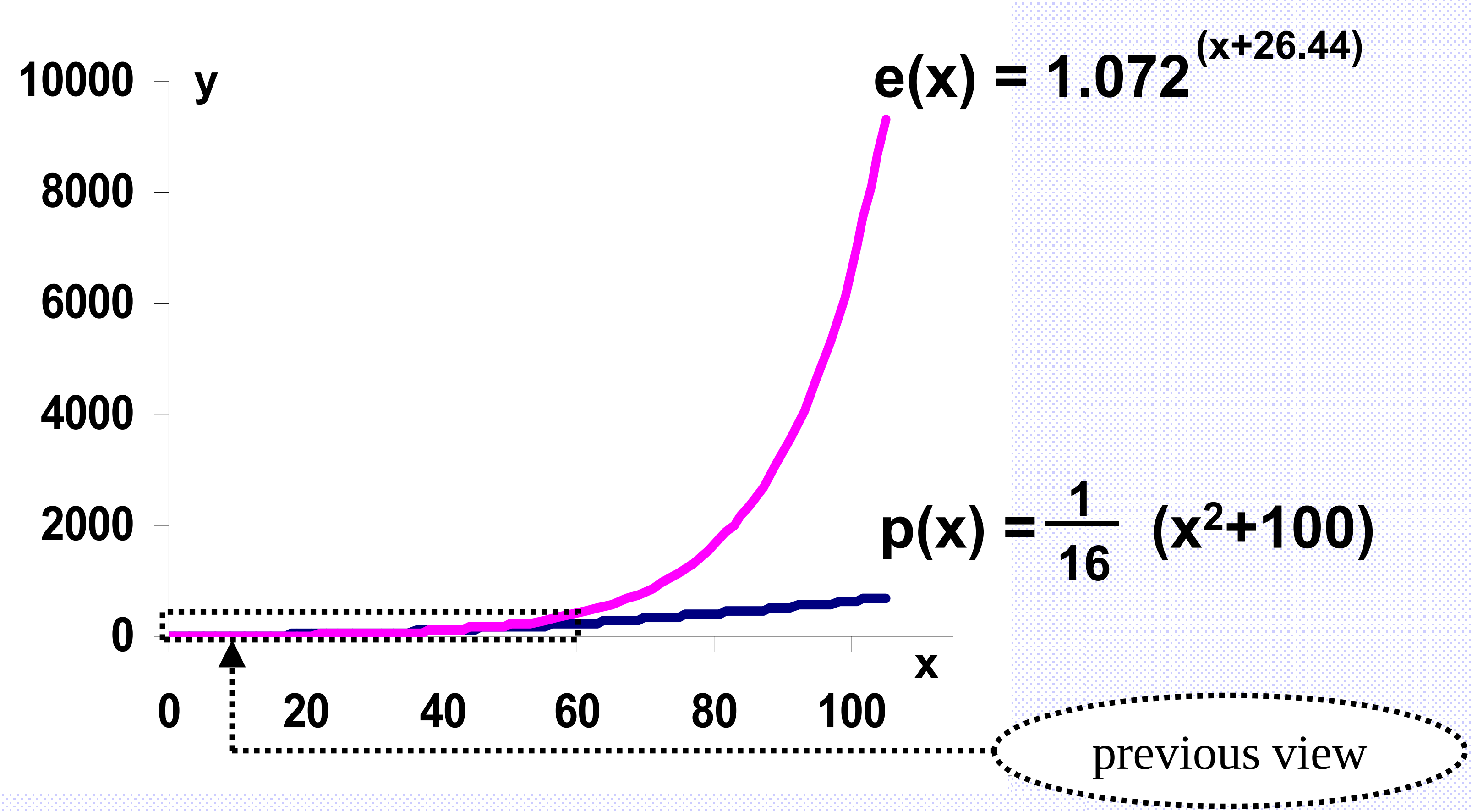


$$e(x) = 1.072^{(x+26.44)}$$

$$p(x) = \frac{1}{16}(x^2+100)$$

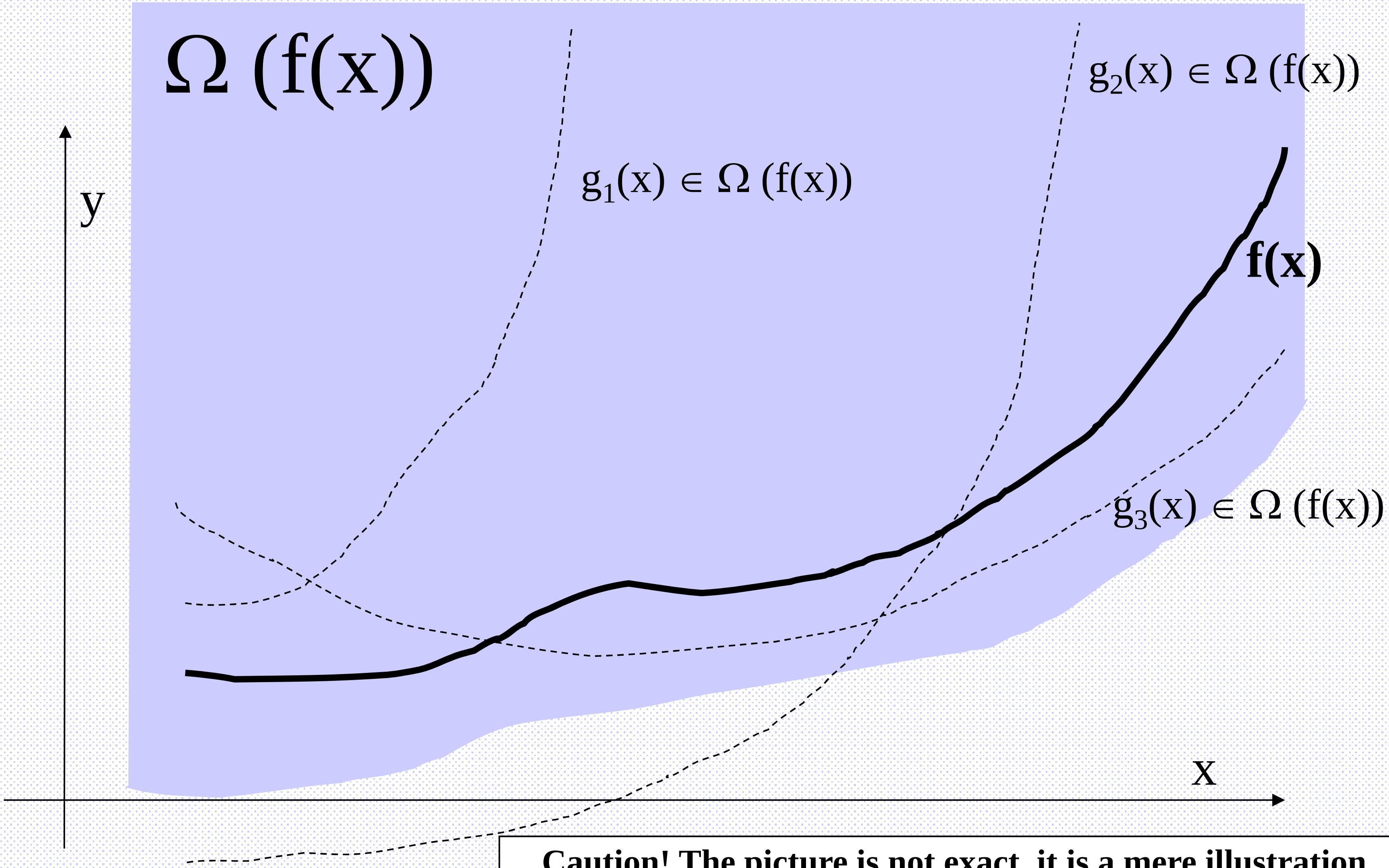
Functions' order of growth

Zoom out! :



etc:… e(1000) = 98431812366054089065476287043 42.9 p(1000) = 62506.25 …

Functions’ order of growth



Caution! The picture is not exact, it is a mere illustration.

Functions' order of growth

Ω(f(x))

Ω Omega

The set Ω(f(x))
contains every function g(x) which from some point x_0 on
(and the position of x_0 is completely arbitrary)

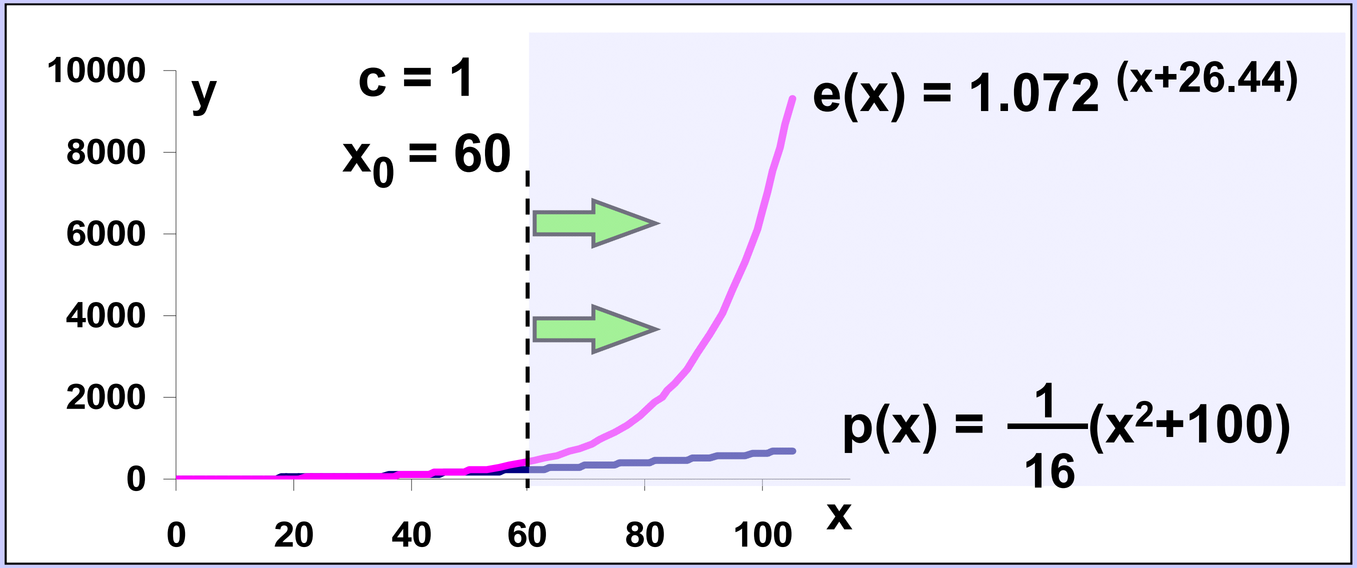
a) – has always bigger value than function f(x) OR

b) – has not bigger value than f(x), however
after being multiplied by some positive constant
(the constant value is arbitrary as well)
has always biggeer value than function f(x).

Thus: if we find some x_0 and c>0 such that
c·g(x) > f(x) everywhere to the right of x_0
(sometimes c=1 is enough), then surely g(x) ∈ Ω(f(x))

Functions' order of growth

Thus: if we find some x_0 and $c>0$ such that $c \cdot g(x) > f(x)$ everywhere to the right of x_0 (sometimes c=1 is enough), then surely $g(x) \in \Omega(f(x))$



$x > 60 \Rightarrow$ e(x) > p(x), i.e. $1.072^{(x+26.44)} > \frac{1}{16}(x^2+100)$

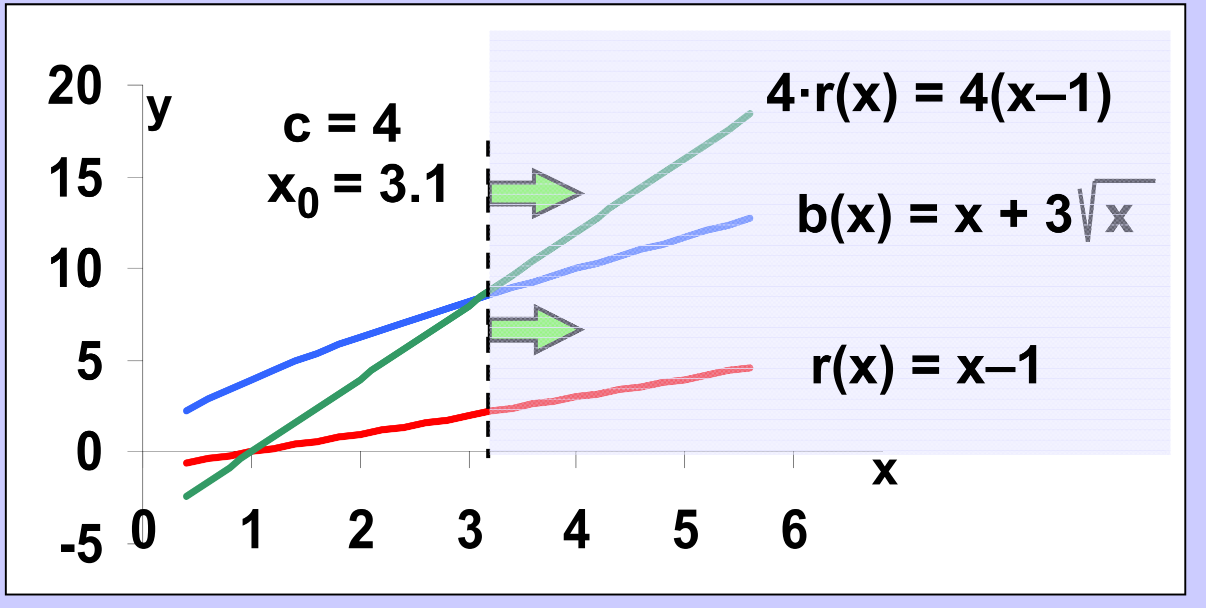
hence holds $\boxed{e(x) \in \Omega(p(x))}$ (check it!)

Functions' order of growth

Thus: if we find some x_0 and $c>0$ such that
$c \cdot g(x) > f(x)$ everywhere to the right of x_0
(sometimes $c=1$ is enough), then surely $g(x) \in \Omega(f(x))$

$b(x) = x + 3\sqrt{x}$

$r(x) = x-1$



$x > 3.1 \quad \Rightarrow \quad 4 \cdot r(x) > b(x)$, i.e. $4(x-1) > x + 3\sqrt{x}$ (check it!)

hence holds $\boxed{r(x) \in \Omega(b(x))}$

Functions' order of growth

Typical examples

$x^2 \in \Omega(x)$	$x^3 \in \Omega(x^2)$	$x^{n+1} \in \Omega(x^n)$
$2^x \in \Omega(x^2)$	$2^x \in \Omega(x^3)$	$2^x \in \Omega(x^{5000})$
$x \in \Omega(\log(x))$	$x\cdot\log(x) \in \Omega(x)$	$x^2 \in \Omega(x\cdot\log(x))$
$2^x \in \Omega(x^{20000})$	$x^{20000} \in \Omega(x)$	$x \in \Omega(1)$

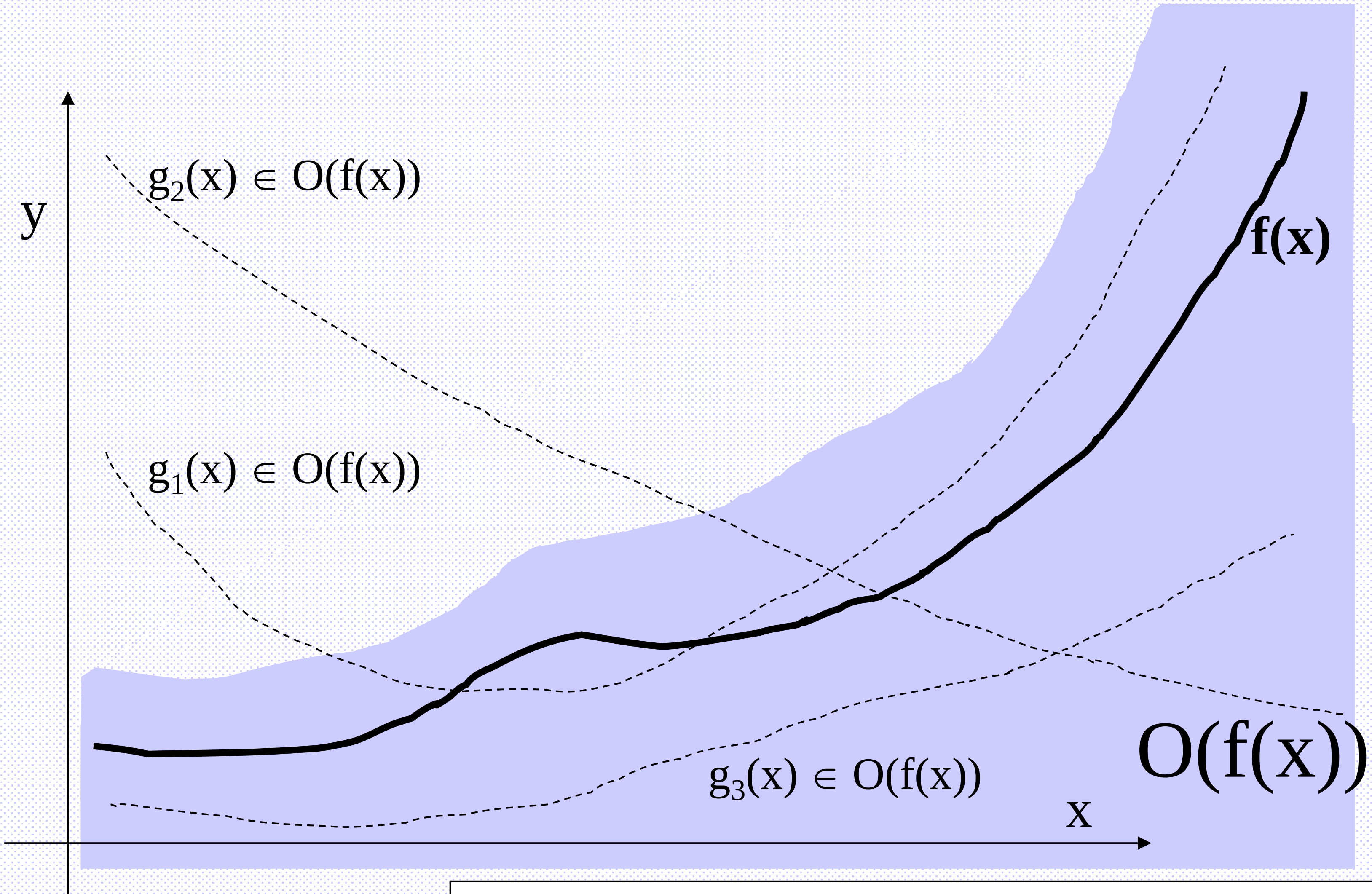
always

$f(x) > 1 \quad \Rightarrow \quad f(x) \in \Omega(1)$

hard to believe

$\sqrt[200\,000]{x} \in \Omega(\log(x)^{200\,000})$

Functions' order of growth



Caution! The picture is not exact, it is a mere illustration.

Functions' order of growth

$O(f(x))$

O Omicron

The set $O(f(x))$
contains each function(x) which from some point x_0 on
(and the position of x_0 is completely arbitrary)

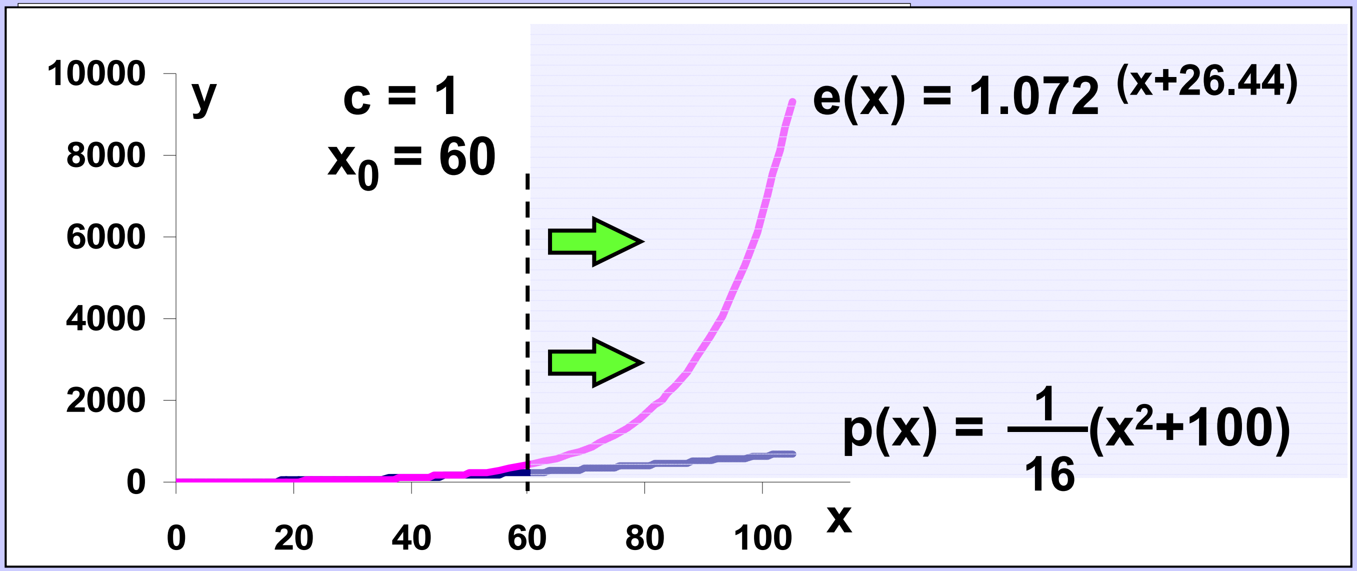
a) – has always smaller value than function $f(x)$

b) – has not smaller value than $f(x)$, however
after being multiplied by some positive constant (< 1 ☺)
(the constant value is arbitrary as well)
has always smaller value than $f(x)$.

Thus: if we find some x_0 and $c>0$ such that
$c \cdot g(x) < f(x)$ everywhere to the right of x_0,
(sometimes $c=1$ suffices) then surely, $g(x) \in O(f(x))$

Functions' order of growth

Thus: if we find some x_0 and $c>0$ such that
$c \cdot g(x) < f(x)$ everywhere to the right of x_0,
(sometimes $c=1$ suffices) then surely, $g(x) \in O(f(x))$



$x > 60 \quad \Rightarrow \quad p(x) < e(x)$, i.e. $\quad \frac{1}{16}(x^2+100) < 1.072^{(x+26.44)}$

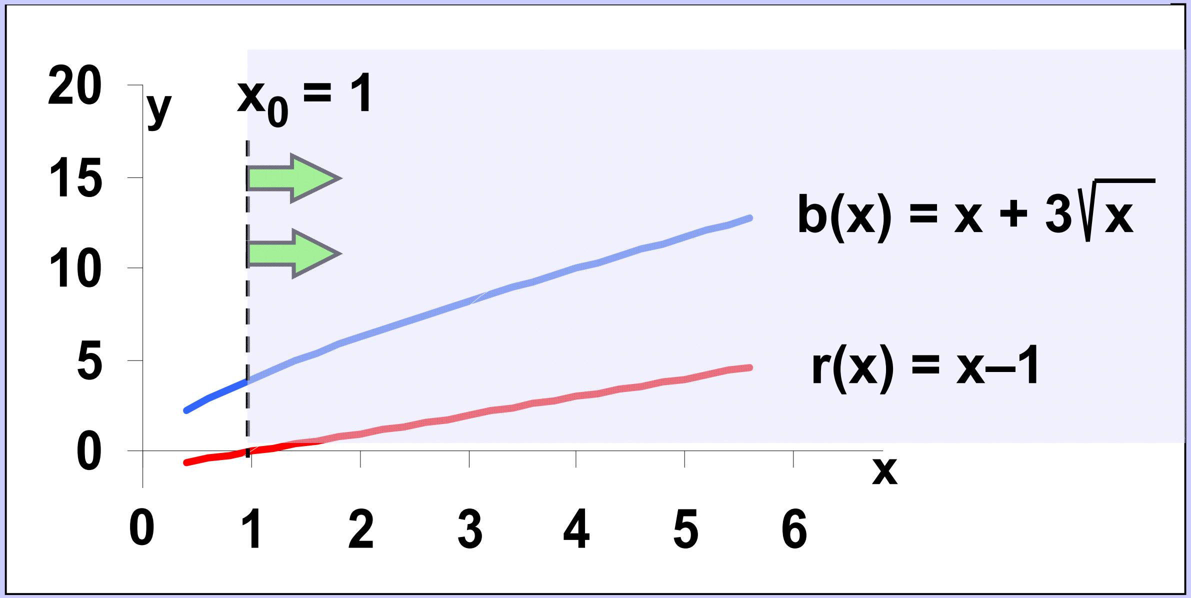
hence holds $\boxed{p(x) \in O(e(x))}$ (check it!)

Functions' order of growth

Thus: if we find some x_0 and $c>0$ such that $c \cdot g(x) < f(x)$ everywhere to the right of x_0, (sometimes c=1 suffices) then surely, $g(x) \in O(f(x))$

$b(x) = x + 3\sqrt{x}$

$r(x) = x - 1$



$x > 1 \quad \Rightarrow \quad r(x) < b(x), \quad$ i.e. $\quad x - 1 < x + 3\sqrt{x}$

hence holds $\boxed{r(x) \in O(b(x))}$

Functions' order of growth

$$f \in \Omega(g) \iff g \in O(f)$$

$x \in O(x^2)$ $x^2 \in O(x^3)$ $x^n \in O(x^{n+1})$

$x^2 \in O(2^x)$ $x^3 \in O(2^x)$ $x^{5000} \in O(2^x)$

$\log(x) \in O(x)$ $x \in O(x\cdot\log(x))$ $x\cdot\log(x) \in O(x^2)$

$x^{20000} \in O(2^x)$ $x \in O(x^{20000})$ $1 \in O(x)$

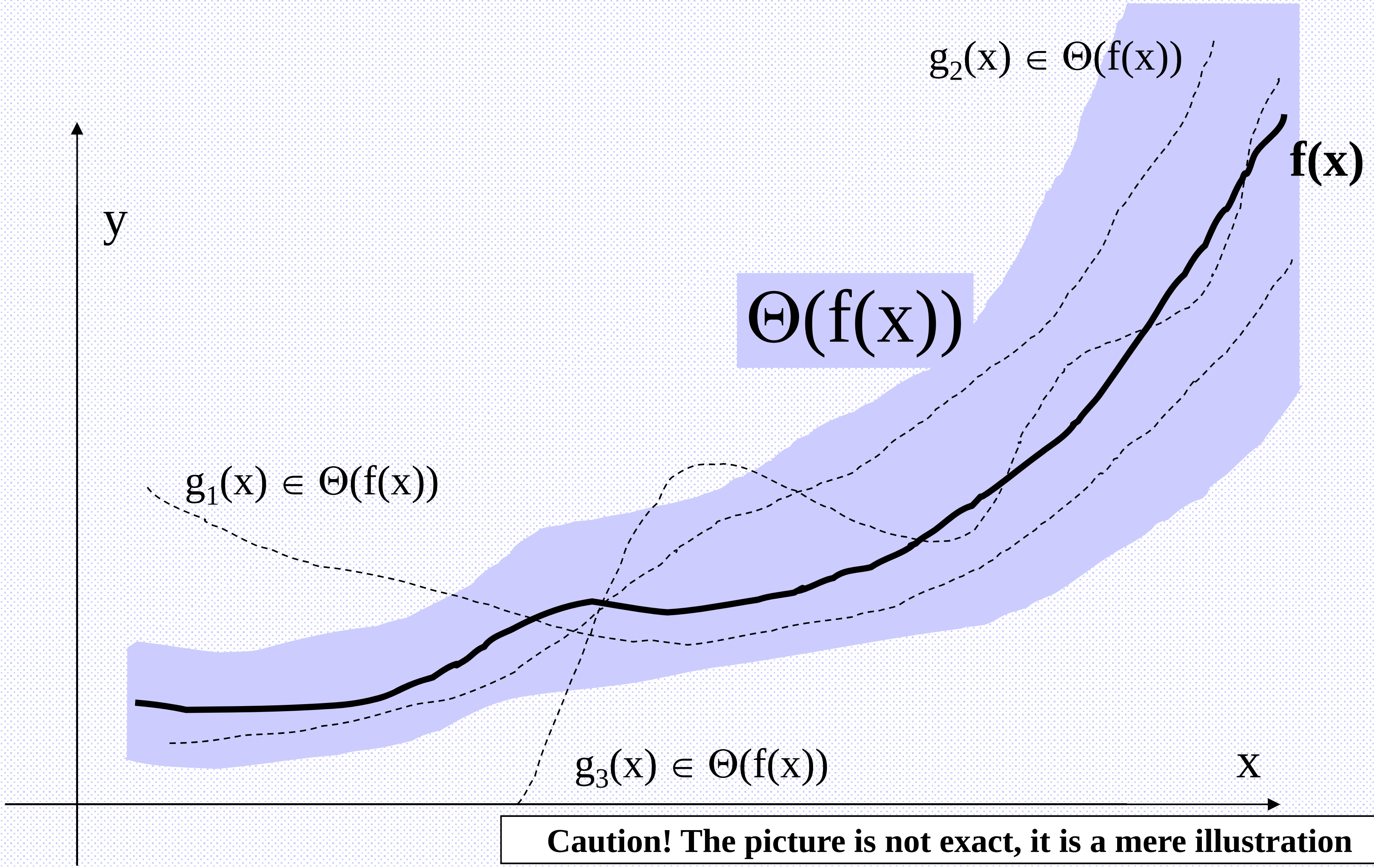
always

$$f(x) > 1 \Rightarrow 1 \in O(f(x))$$

hard to believe

$$\log(x)^{200\,000} \in O(\sqrt[200\,000]{x})$$

Functions' order of growth



Caution! The picture is not exact, it is a mere illustration

Functions' order of growth

$$\Theta(f(x)) = \Omega(f(x)) \cap O(f(x))$$

Θ Theta

The set $\Theta(f(x))$ contains every function $g(x)$

which belongs to both $\Omega(f(x))$ and $O(f(x))$.

$$f(x) \in \Theta(g(x)) \quad \Leftrightarrow \quad g(x) \in \Theta(f(x))$$

Functions' order of growth

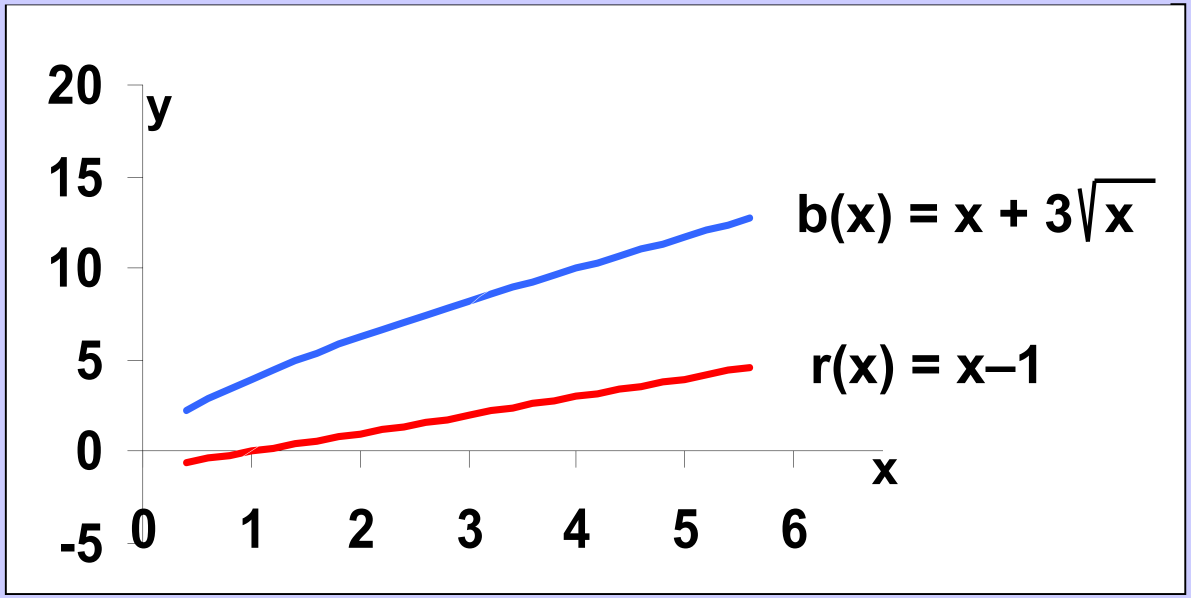
$$f(x) \in \Theta(g(x)) \quad \Leftrightarrow \quad g(x) \in \Theta(f(x))$$

$b(x) = x + 3\sqrt{x}$

$r(x) = x - 1$

$r(x) \in \Omega(b(x))$

$r(x) \in O(b(x))$



$r(x) \in \Theta(b(x))$ $\quad$ $b(x) \in \Theta(r(x))$

Functions' order of growth

Rules

1. $(a > 0) \Leftrightarrow \Theta(f(x)) = \Theta(a \cdot f(x))$

2. $g(x) \in O(f(x)) \Leftrightarrow \Theta(f(x)) = \Theta(f(x) + g(x))$

In words

1. Multiplication by positive constant does not affect belonging to $\Theta(f(x))$.
2. Addition or substraction of a „smaller“ function does not affect belonging to $\Theta(f(x))$.

Examples

$$1.8x + 600 \cdot \log_2(x) \in \Theta(x)$$

$$x^3 + 7x^{1/2} + 5(\log_2(x))^4 \in \Theta(x^3)$$

$$13 \cdot 3^x + 9x^{12} + 42x^{-4} + 29 \in \Theta(3^x)$$

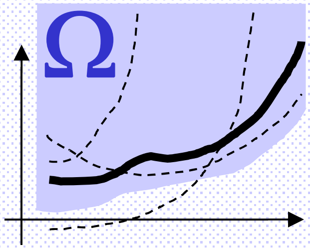
$$4 \cdot 2^n + 3 \cdot 2^{n-1} + 5 \cdot 2^{n/2} \in \Theta(2^n)$$

$$0.1x^5 + 200x^4 + 7x^2 - 3 \in \Theta(x^5)$$

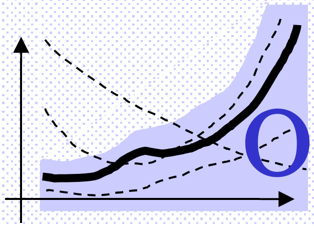
–”– $\in O(x^5)$

–”– $\in \Omega(x^5)$

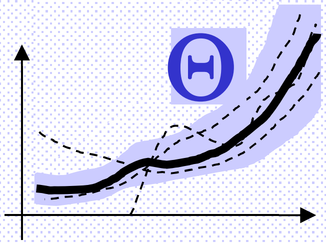
Functions' order of growth



$$\Omega(f(x)) = \{ g(x) \ ; \exists x_0>0, c>0 \ \forall x>x_0: c \cdot f(x) < g(x) \}$$



$$O(f(x)) = \{ g(x) \ ; \exists x_0>0, c>0 \ \forall x>x_0: g(x) < c \cdot f(x) \}$$



$$\Theta(f(x)) = \{ g(x) \ ; \exists x_0>0, c_1>0, c_2>0 \ \forall x>x_0: c_1 \cdot f(x) < g(x) < c_2 \cdot f(x) \}$$

Caution! The pictures are not exact, they are mere illustration.

Functions' order of growth

Comparing the speed of growth of functions

Function f(x) grows asymptotically faster than function g(x) when

$$f(x) \in \Omega(g(x)) \;\;\&\;\; f(x) \notin \Theta(g(x))$$

Be careful!

Comparing the speed of algorithms

Algorithm A is asymptotically slower then algorithm B when

$$f_A(n) \in \Omega(f_B(n)) \;\;\&\;\; f_A(n) \notin \Theta(f_B(n)),$$

where $f_A(n)$, resp. $f_B(n)$ is a function determining the number of operations executed by algorithm A, resp. B when they process data of size n.

Functions' order of growth

Order of growth of a function

Order of growth of function f
is "the most simple" function g, for which holds
$g(x) \in \Theta(f(x))$

Manipulation

The order of growth is mostly obtained by dropping

1. additive members of "slower or equal" rate of growth,
2. multiplicative constants.

Examples

$ff(n) = 4 \cdot 2^n + 3 \cdot 2^{n-1} + 5 \cdot 2^{n/2} \in \Theta(2^n)$ order of growth is 2^n

$hh(x) = x + \log_2(x) - \sqrt{x} \in \Theta(x)$ order of growth is x

Asymptotic complexity

Asymptotic complexity of an algorithm

Asymptotic complexity of algorithm A
is the order of growth of the function f(n) which chracterizes
maximum number of elementary operations
which algoritm A performs when it processes any data of size n.

We suppose that the data are the most "difficult" ones.

(size of data = the total number of data elements)

Mostly it makes no difference if we consider

A) total of all elementary operations,
B) total of all elementary operations on data,
C) total of tests on data.

The asymptotic complexity is usually the same.

Asymptotic complexity

Asymptotic complexity of the introductory examples

Searching for min and max in an array.
Asymptotic complexity is $\Theta(\underline{n})$ in both cases.

Checking how many elements are equal to sum of an array.
Asymptotic complexity of the SLOW solution is $\Theta(\underline{n^2})$.

Asymptotic complexity of the FAST solution is $\Theta(\underline{n})$.

Assuming both arrays are of length $\underline{n}$.

Asymptotic complexity of linear search in a sorted array is $O(\underline{n})$.

Asymptotic complexity of binary search in a sorted array is $O(\underline{\log(n)})$.

Assuming the array is of length $\underline{n}$.

Asymptotic complexity

Conventions

Simplification

Usually the term „algorithm complexity“ is interpreted as „asymptotic complexity of the algorithm“.

Confusion

Usually they do not say f(x) <u>belongs to</u> $\Theta(g(x))$,

but rather f(x) <u>is</u> $\Theta(g(x))$.

And they mark it accordingly $f(x) = \Theta(g(x))$

instead of $f(x) \in \Theta(g(x))$.

The same convention holds for O and Ω.

But they think of it in the original meaning defined above.

Asymptotic complexity

 $\in \Theta($  $)$

 $\in \Theta($  $)$

 $\in \Omega($  $)$

 $\in O($  $)$

The complexity

of different algorithms

varies