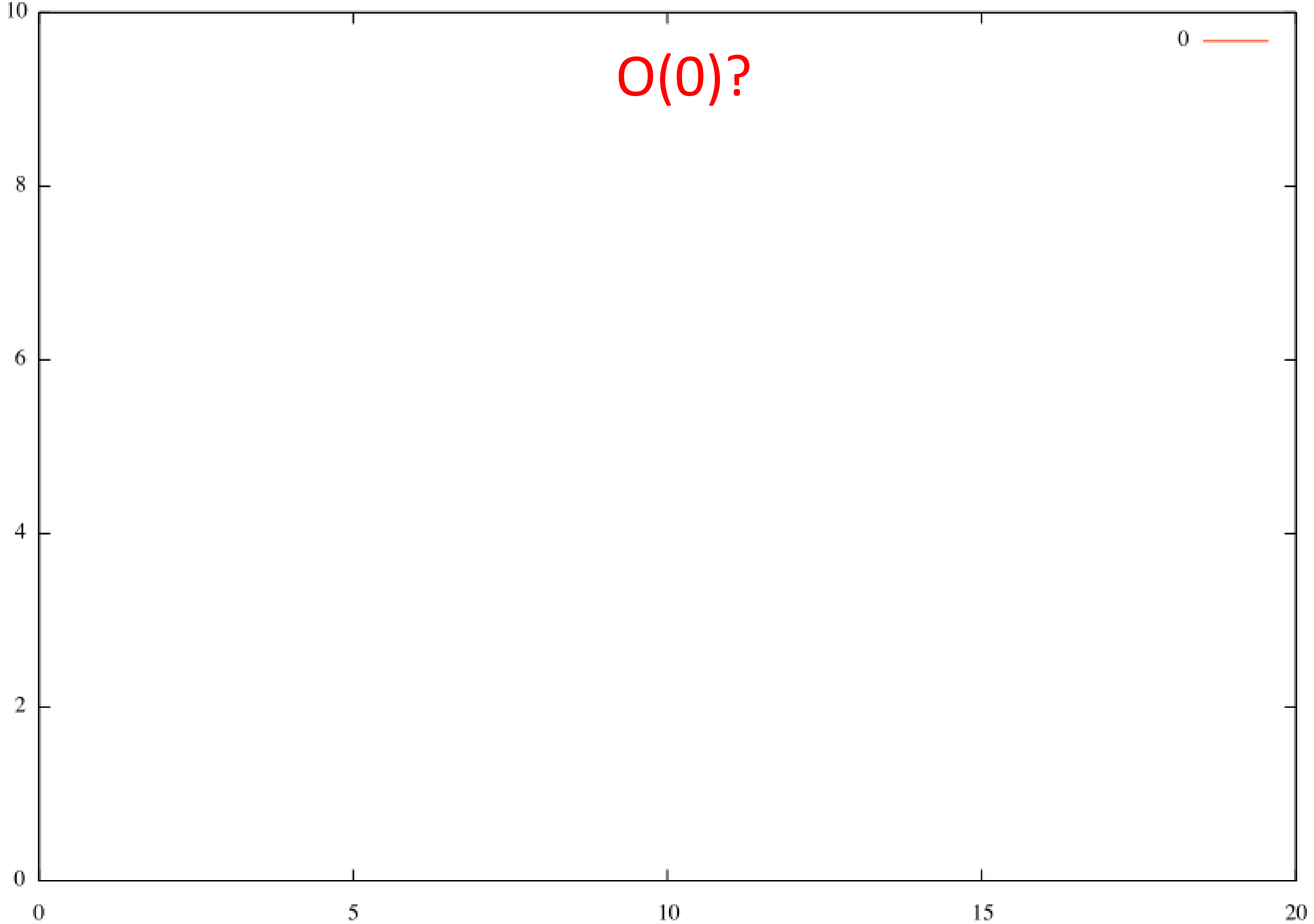
Algorithms

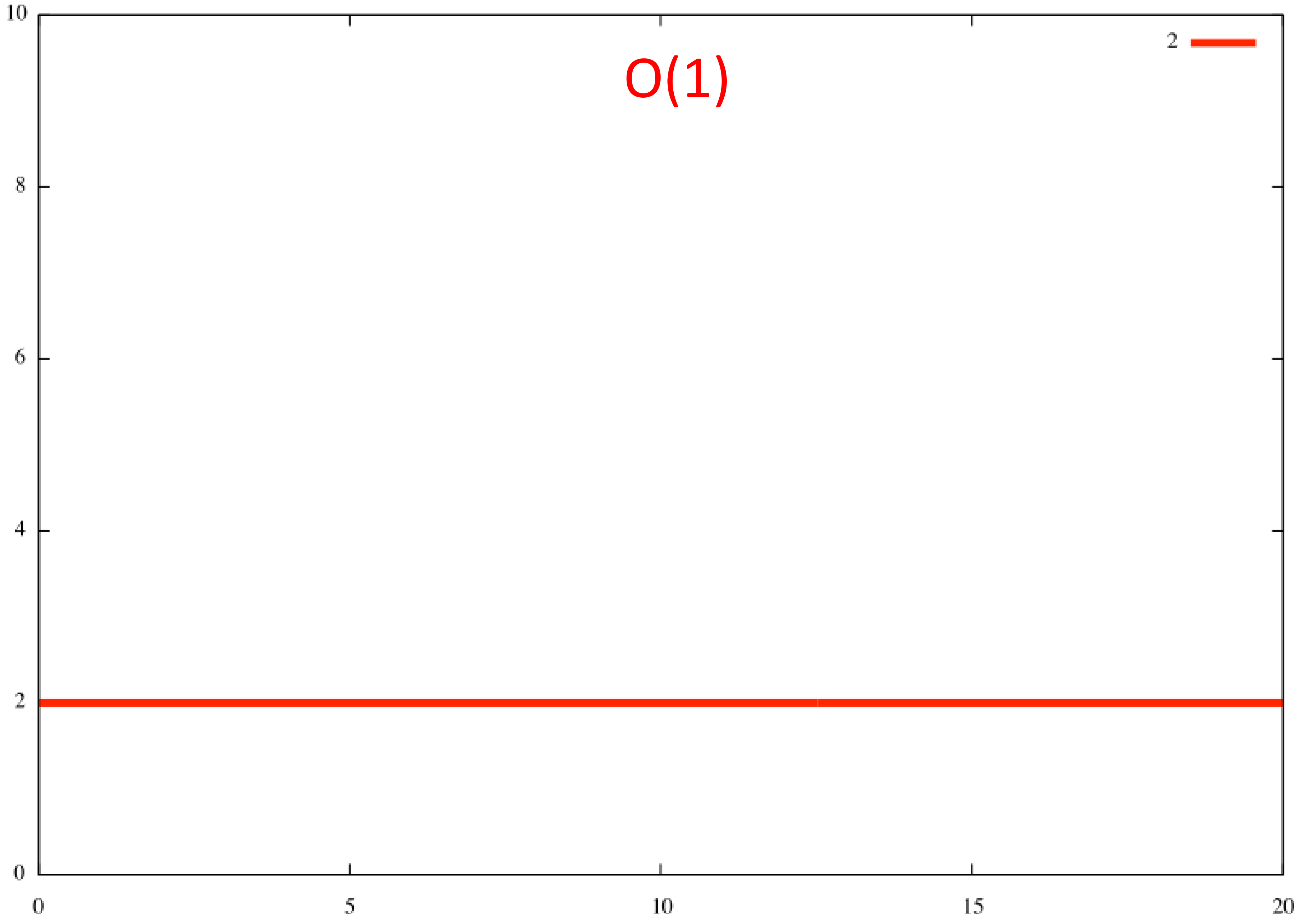
Classifying algorithms by the rate of growth

Lecture 10 by *Marina Barsky*

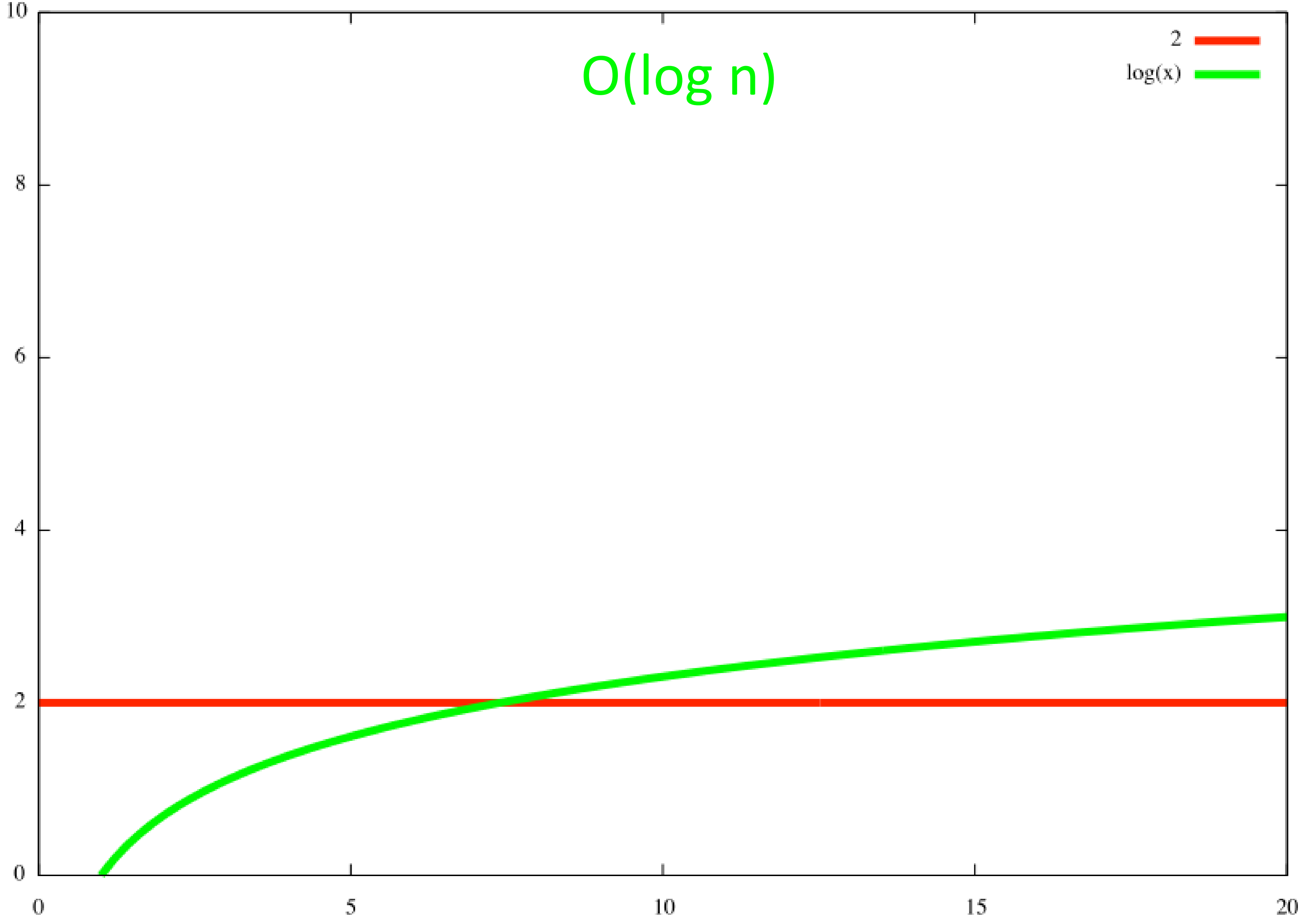
Classifying algorithms by the rate of growth



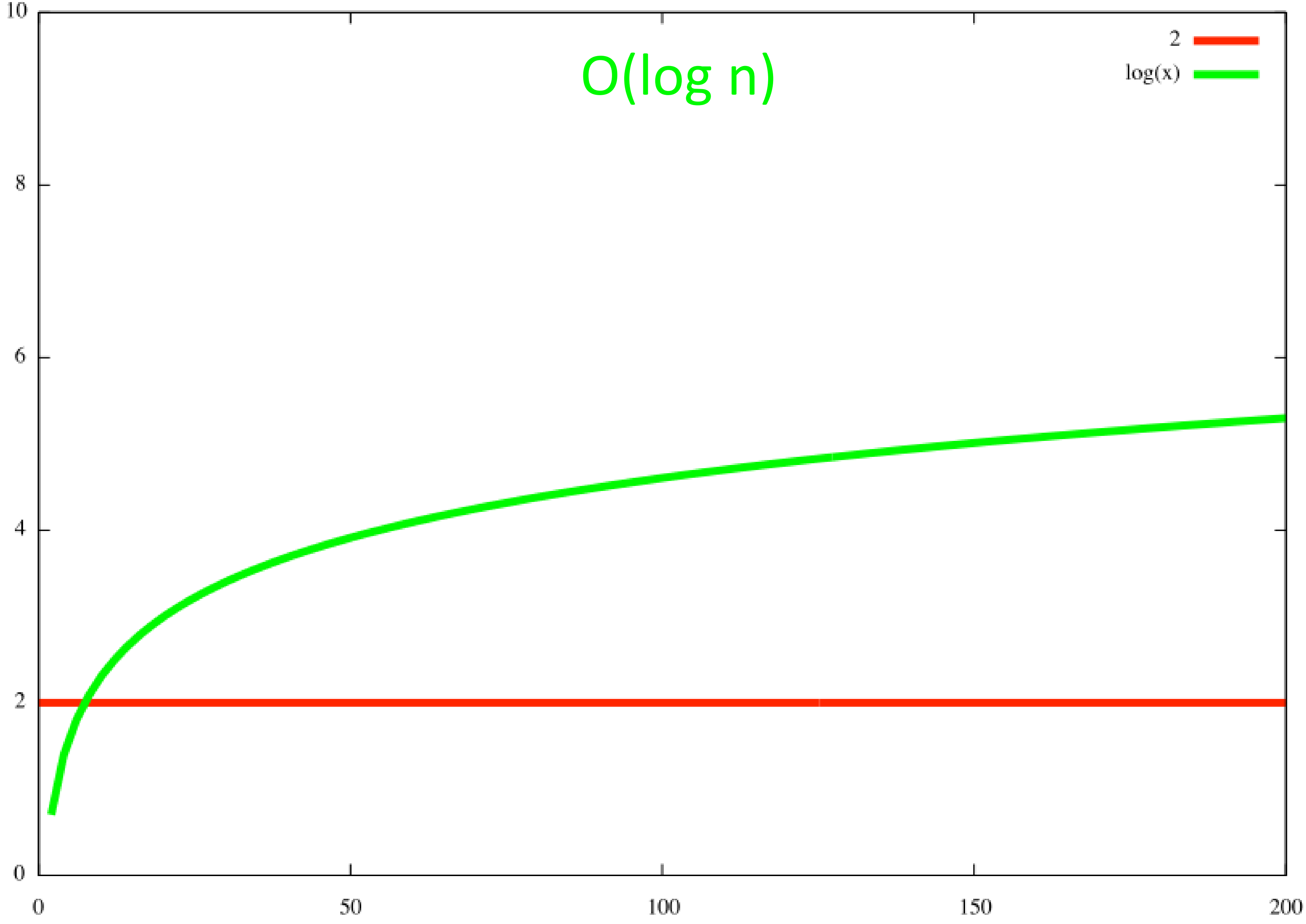
0
O(0)?
10
8
6
4
2
0
0
5
10
15
20

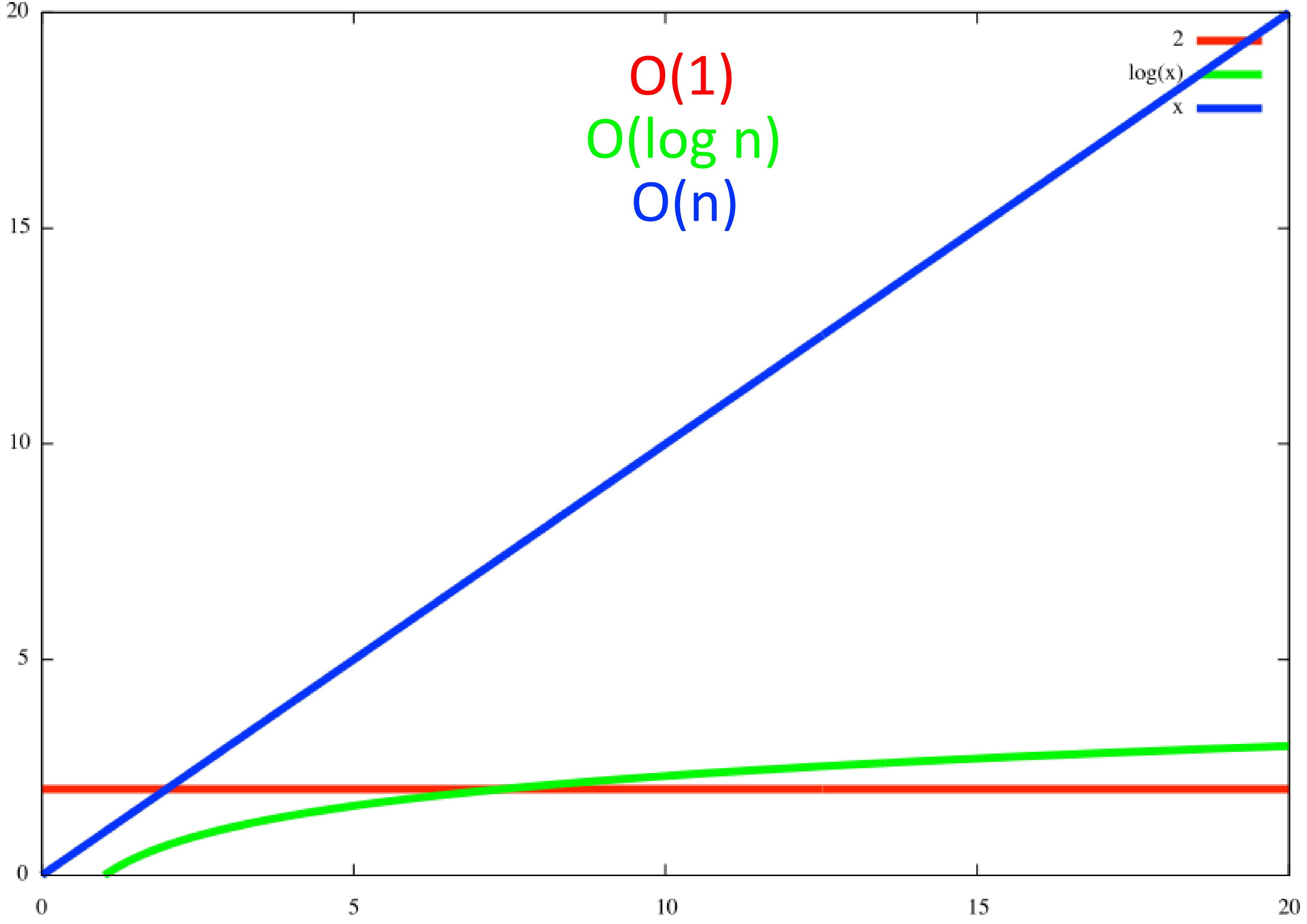


O(1)
2
10
8
6
4
2
0
0
5
10
15
20

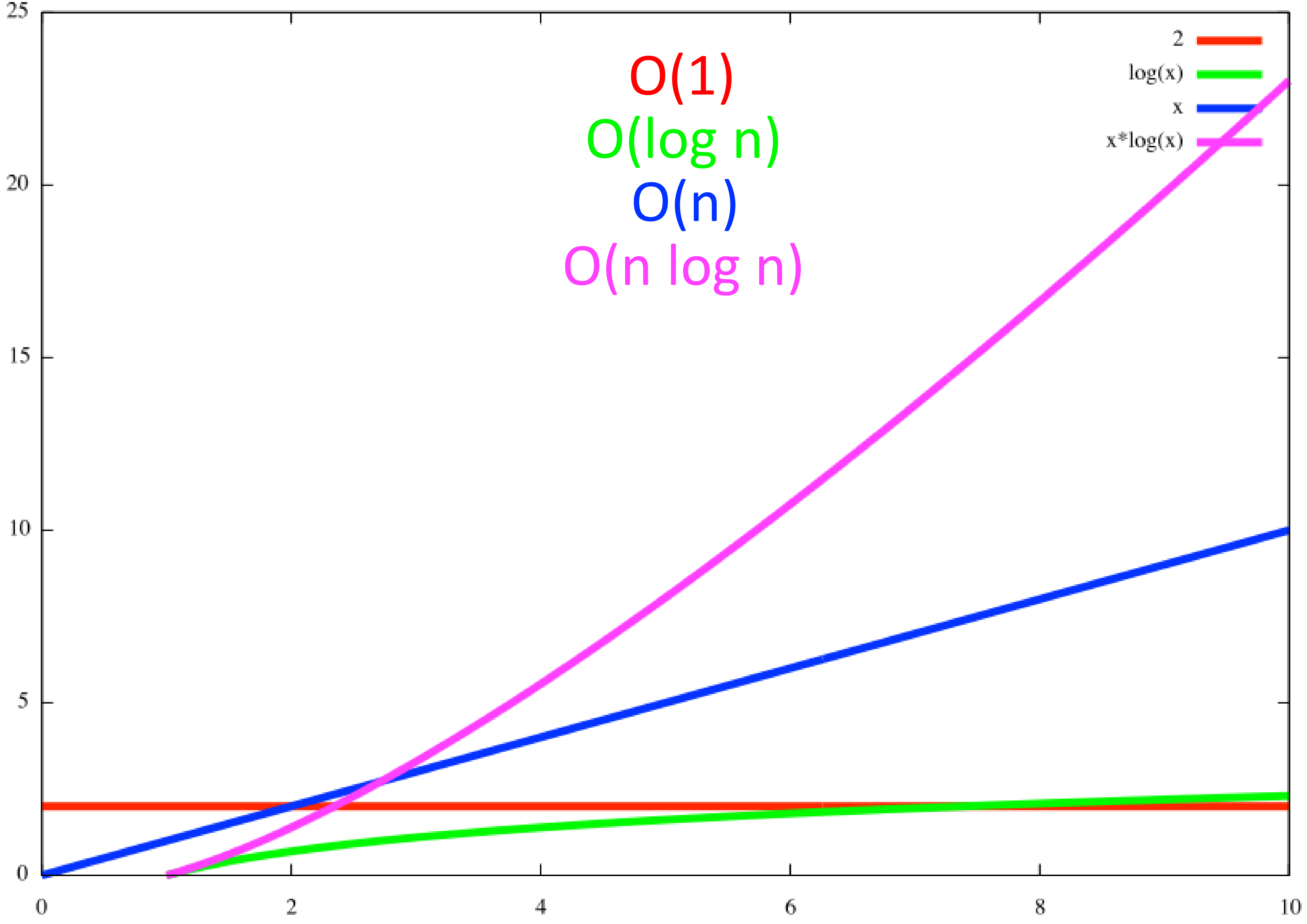


O(log n)
2
log(x)
10
8
6
4
2
0
0
5
10
15
20

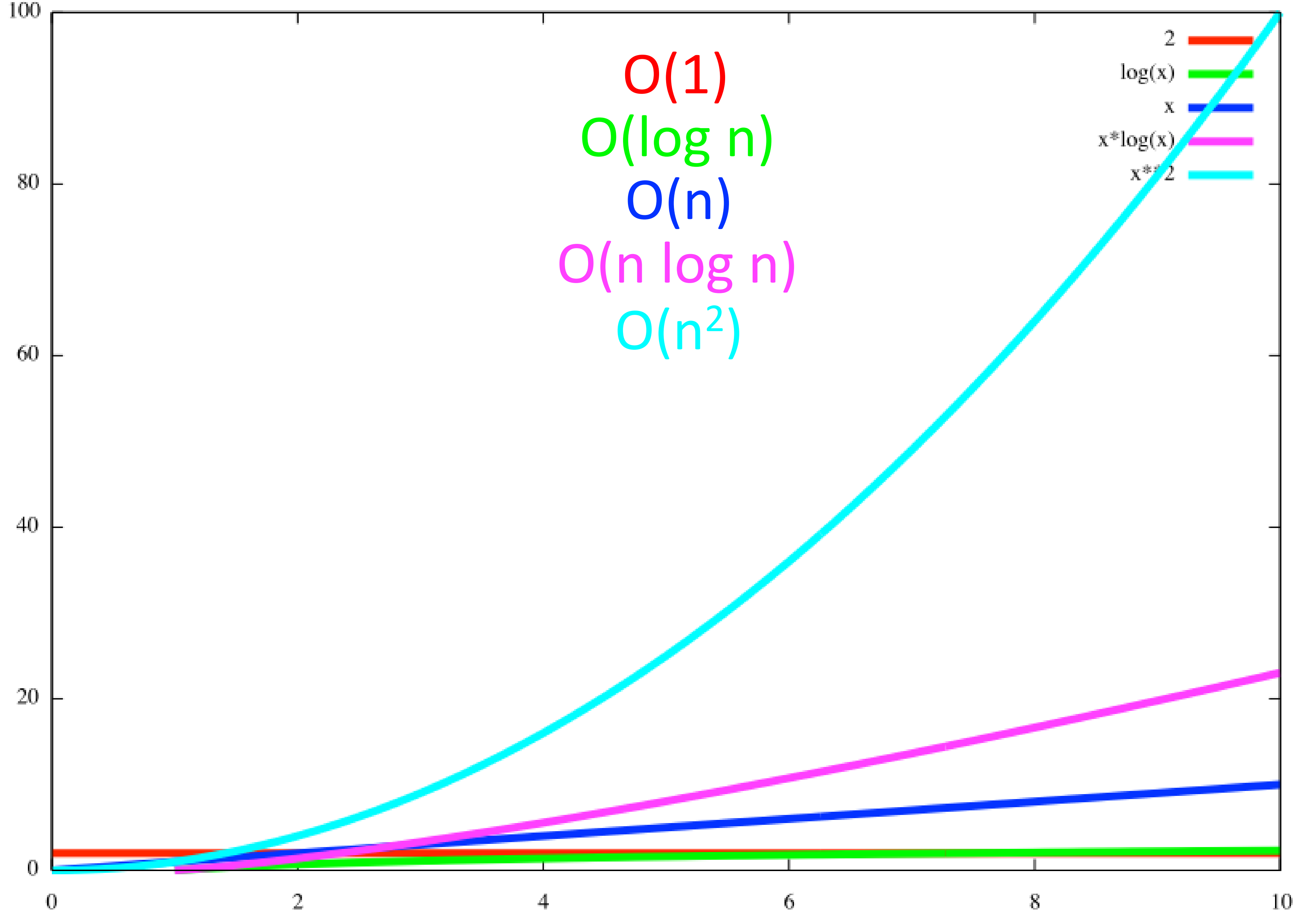

O(log n)
2
log(x)
10
8
6
4
2
0
0
50
100
150
200

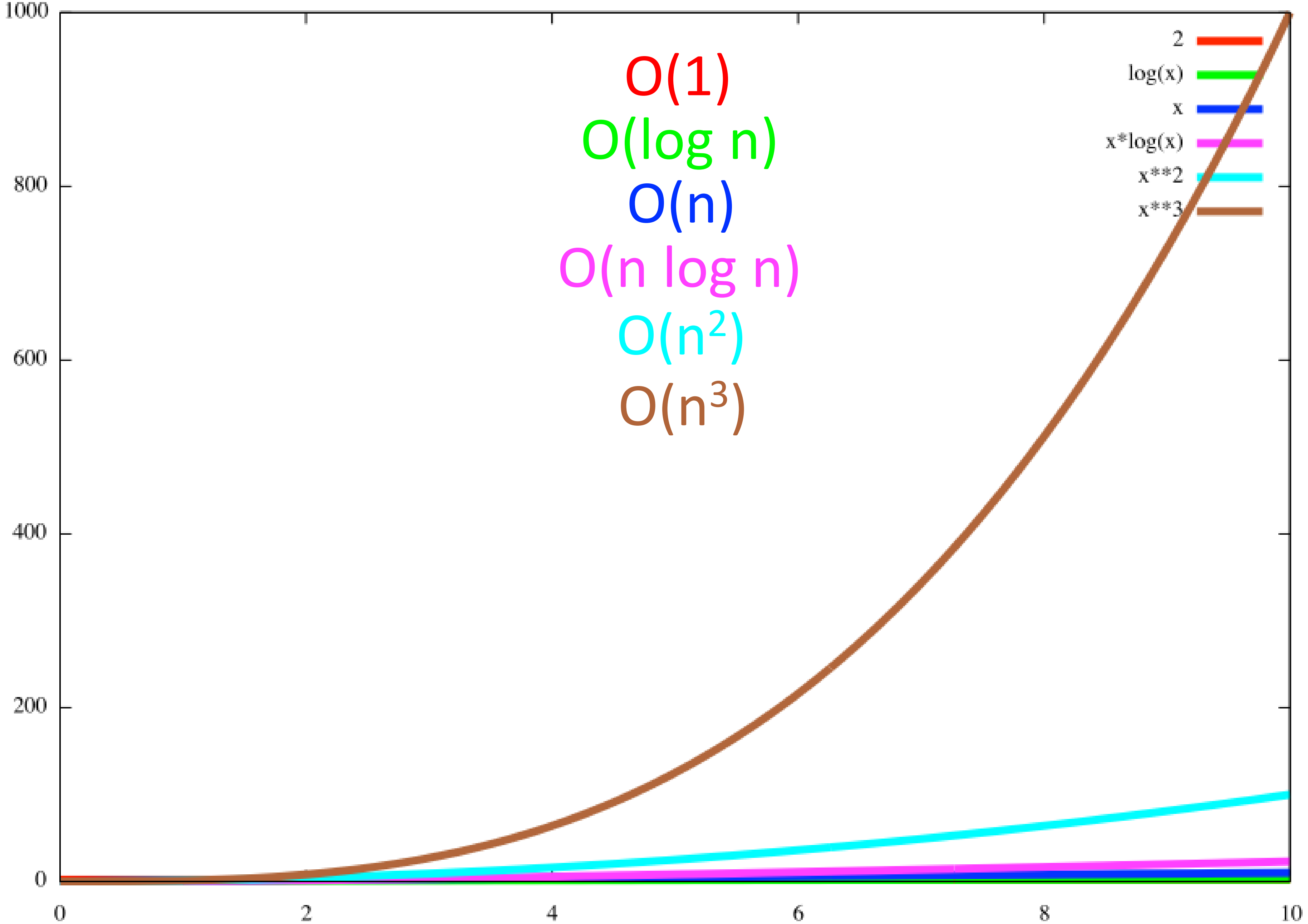


O(1)
O(log n)
O(n)
2
log(x)
x
0
5
10
15
20
0
5
10
15
20

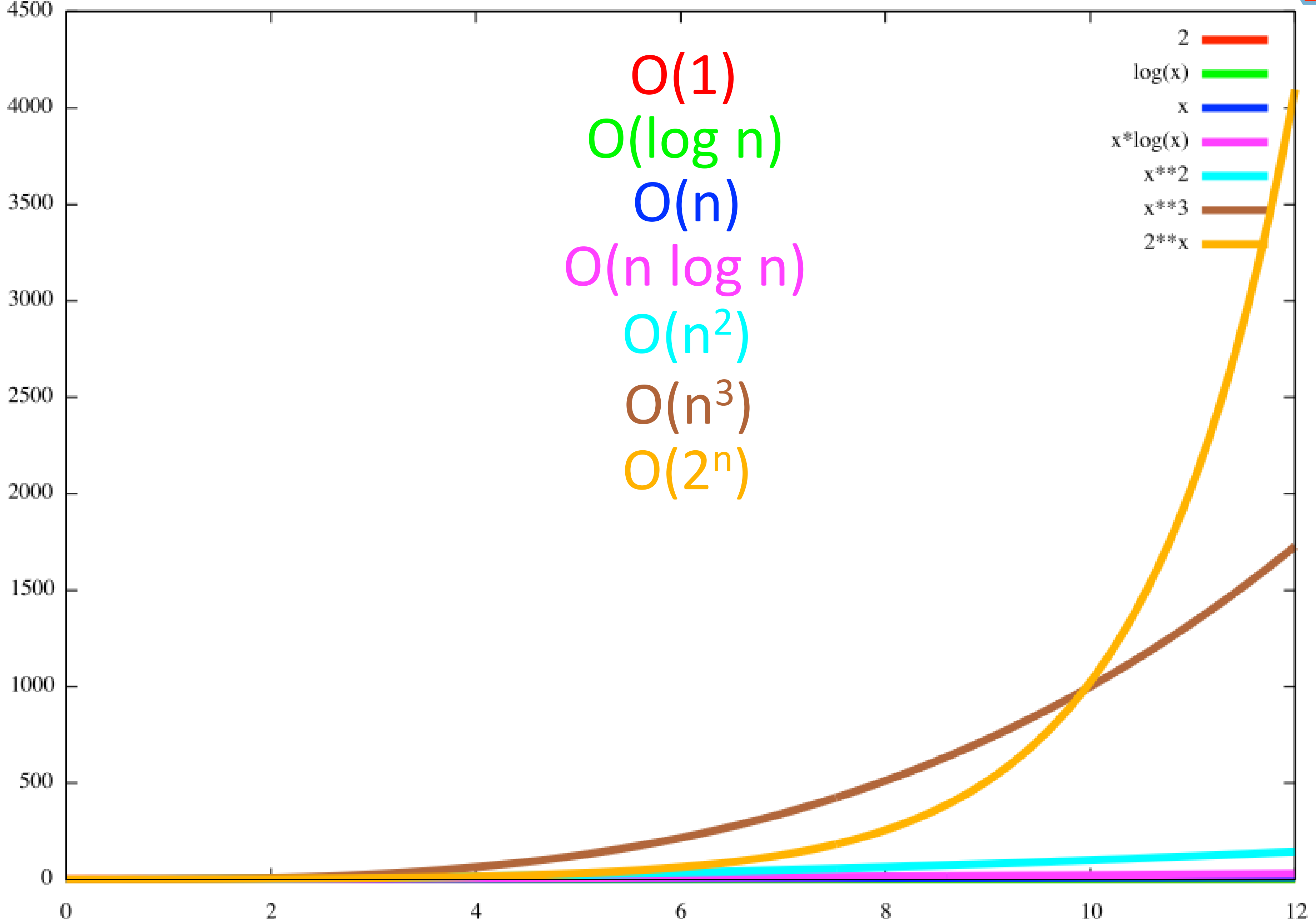


O(1)
O(log n)
O(n)
O(n log n)
2
log(x)
x
x*log(x)
0
5
10
15
20
25
0
2
4
6
8
10


O(1)
O(log n)
O(n)
O(n log n)
O(n²)
2
log(x)
x
x*log(x)
x**2
0
20
40
60
80
100
0
2
4
6
8
10



O(1)
O(log n)
O(n)
O(n log n)
O(n2)
O(n3)
2
log(x)
x
x*log(x)
x**2
x**3
1000
800
600
400
200
0
0
2
4
6
8
10

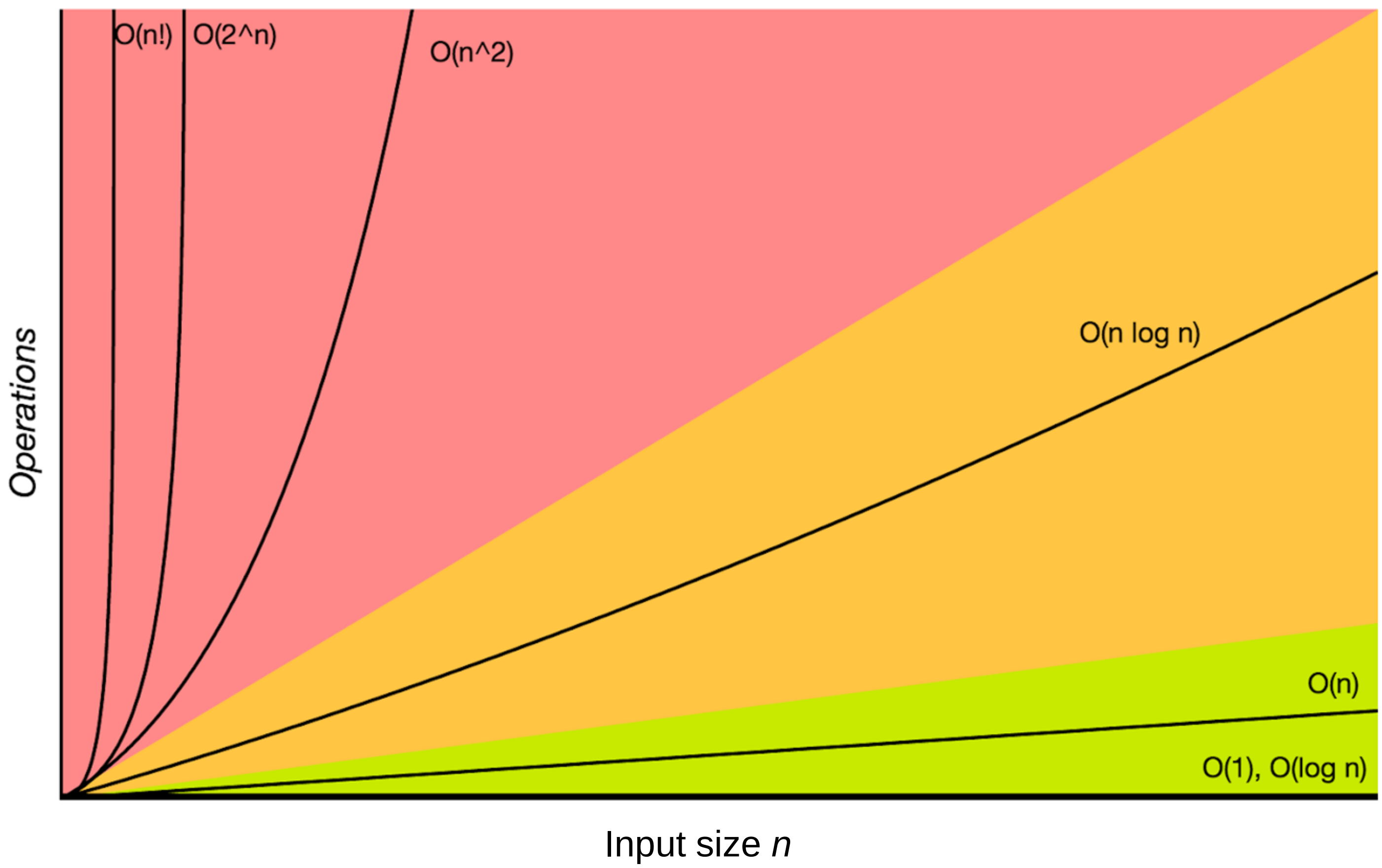


O(1)
O(log n)
O(n)
O(n log n)
O(n2)
O(n3)
O(2n)
2
log(x)
x
x*log(x)
x**2
x**3
2**x
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
2
4
6
8
10
12

Examples

- **O(1)**
  - Getting the length of a given array
  - Getting the i-th element from *ArrayList*
- **O(n)**
  - Min/Max value in an array
  - Search for something in an unsorted list
- **O(n^2)**
  - Finding closest pair of points in a plane

Algorithms: practical and impractical



What does it mean in practice

Assuming n=1,000 and 1ms per operation

Name	Big O	Time to process	Max n per day
Constant	O(1)	1 ms	
Logarithmic	O(log n)	9.9 ms	
Linear	O(n)	1 s	86,400,000
n log n	O(n log n)	9.9 s	3,943,234
Quadratic	$O(n^2)$	16.67 min	9,295
Cubic	$O(n^3)$	11.57 days	442
Exponential	$O(2^n)$	$3.395*10^{290}$ years	26
Factorial	O(n!)	???	11

n bytes	log n	n	n^2	2^n
10 B	1	10	100	$\sim 1*10^3$
100 B	2	100	10000	$\sim 1*10^{30}$
1 KB	3	1,000	1000000	$\sim 1*10^{300}$
10 KB	4	10,000	100000000	$\sim 1*10^{3000}$
100 KB	5	100,000	10000000000	$\sim 1*10^{30,000}$
1 MB	6	1,000,000	1.00E+12	$\sim 1*10^{300,000}$
10 MB	7	10,000,000	1.00E+14	n/a
100 MB	8	100,000,000	1.00E+16	n/a
1 GB	9	1,000,000,000	1.00E+18	n/a
10 GB	10	10,000,000,000	1.00E+20	n/a
100 GB	11	100,000,000,000	1.00E+22	n/a
1 TB	12	1,000,000,000,000	1.00E+24	n/a

CPU with a clock speed of 2 gigahertz (GHz) can carry out two thousand million (**$2*10^9$**) cycles (operations) **per second**.

- Algorithm which runs in **$O(2^n)$** time will process **1 KB** of input in **$\sim 10^{300}$ years** (more than 100 millennia)
- Processing **1 GB** of input will take **<0.001 ms** by O(log n) algorithm, **< 1 sec** by **O(n)** algorithm, and **>32 years** by **$O(n^2)$** algorithm

Complexity of sorting

Sorting 1

```
void sorting1 (array A)
   i = 1
   while i < length(A)
      j = i

      while j > 0 and A[j-1] > A[j]
         swap A[j] and A[j-1]
         j = j - 1

      i = i + 1
```

A. O(n)
B. $O(n^2)$
C. $O(n^3)$
D. None of the above

Sorting 1 is a…

```
void sorting1 (array A)
  i = 1
  while i < length(A)
    j = i

    while j > 0 and A[j-1] > A[j]
      swap A[j] and A[j-1]
      j = j - 1

    i = i + 1
```

A. Bubble sort
B. Insertion sort
C. Selection sort
D. None of the above

Sorting 2

```
void sorting2 (array A)
   n = length(A)
   swapped = false
   do:
      for i from 0 to n-1
         if A[i-1] > A[i]:
            swap A[i-1] and A[i]
            swapped = true
      n = n - 1
   while (swapped)
```

A. O(n)
B. $O(n^2)$
C. $O(n^3)$
D. None of the above

Sorting 2 is a...

```
void sorting2 (array A)
   n = length(A)
   swapped = false
   do:
      for i from 0 to n-1
         if A[i-1] > A[i]:
            swap A[i-1] and A[i]
            swapped = true
      n = n - 1
   while (swapped)
```

A. Bubble sort

B. Insertion sort

C. Selection sort

D. None of the above

Back to basic Data Structures

Complexity of operations on Arrays and Linked Lists

ArrayList and LinkedList: algorithms

- Read:
  - get (index i)
  - indexOf (Object o)

- Edit:
  - add()
  - remove()

Running time of common operations for ArrayList and LinkedList

Operation	**ArrayList**	**LinkedList**
Get i-th element		
Search for an element (indexOf)		
Add new element at the end		
Add element at position *i*		
Remove from the end		
Remove from position *i*		
Resize when full		

Running time of common operations for ArrayList and LinkedList

Operation	ArrayList	LinkedList
Get i-th element	O(1)	O(n)
Search for an element (indexOf)	O(n)	O(n)
Add new element at the end	O(1) O(n) if need to resize	O(n) O(1) with tail pointer
Add element at position *i*	O(n)	Traverse in O(n) then O(1)
Remove from the end	O(1)	O(1) with tail pointer
Remove from position *i*	O(n)	Traverse in O(n) then O(1)
Resize when full	O(n)	n/a: never full

Knowing that worst-case performance of the *add()* method of ArrayLists is O(n), what is the time complexity of the following loop?

```
void addAll(int n) {
   ArrayList list;

   for (int i = 0; i<n; i++){
      list.add(i);
   }
}
```

A. O (n^2)
B. O (n)
C. O (1)
D. None of the above

Resizing arrays: Amortized analysis

Sometimes, looking at the individual worst-case may be too severe.
We may want to know the total worst-case cost for a sequence of operations.

- In dynamic arrays we only resize every so often.
- Many O(1) operations are followed by an O(n) operation.
- What is the total cost of inserting n elements? $O(n^2)$?

Definition

Amortized cost: Given a sequence of n operations, the amortized cost of each operation is:

$$\frac{\text{Cost}\ (n\ \text{operations})}{n}$$

Dynamic arrays: amortized cost of *add*

Intuition:

- Say we originally have *k* elements in the Array List, and the list is half-full
- Now we can add another *k* elements, each in time O(1) – in total k*O(1) = O(k) steps

- Now we need to resize by copying 2*k* elements in time O(2*k*)=O(*k*)

So in total adding *k* new elements takes O(*k*) + O(*k*) = O(*k*) which is O(*k*)/*k* = O(1) amortized cost per single *add*

Aggregate method: cost of *n* calls to *add*

- Let's start with array of size 1
- If we choose the strategy of doubling the size of the array on resizing, then during the insertion of *n* elements we will double and copy in total 1 + 2 + 4 + 8 + …*n*/2 elements
- In total we will perform copy log n times

$$1 + 1\times2 + 1\times2\times2 + 1\times2\times2\times2 + \ldots\ 1\times2^{\log n} =$$
$$1\times2^0 + 1\times2^1 + 1\times2^2 + 1\times2^3 + \ldots\ 1\times2^{\log n}$$

What do we see here?

Aggregate method: cost of *n* calls to *add*

$1\times2^0 + 1\times2^1 + 1\times2^2 + 1\times2^3 + \ldots\ 1\times2^{\log n}$

- This is a sum of geometric series with $a_0=1$, $d=2$, and total of $k=\log n$ elements

- The sum of the first k elements of the geometric series:

  $\text{Sum} = a_0(d^k - 1)/(d - 1)$

- For our case it is:

  $2^k - 1$, and $k = \log n$

  and $2^{\log n} = n$

Aggregate method: cost of *n* calls to *add*

$1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + \ldots 1 \times 2^{\log n}$

- This sum is $O(2^{\log n}) = O(n)$
- Thus the cost of n*add() is O(n), which is O(1) per add

Corollary:

The amortized cost of *add* in dynamic array is O(1)