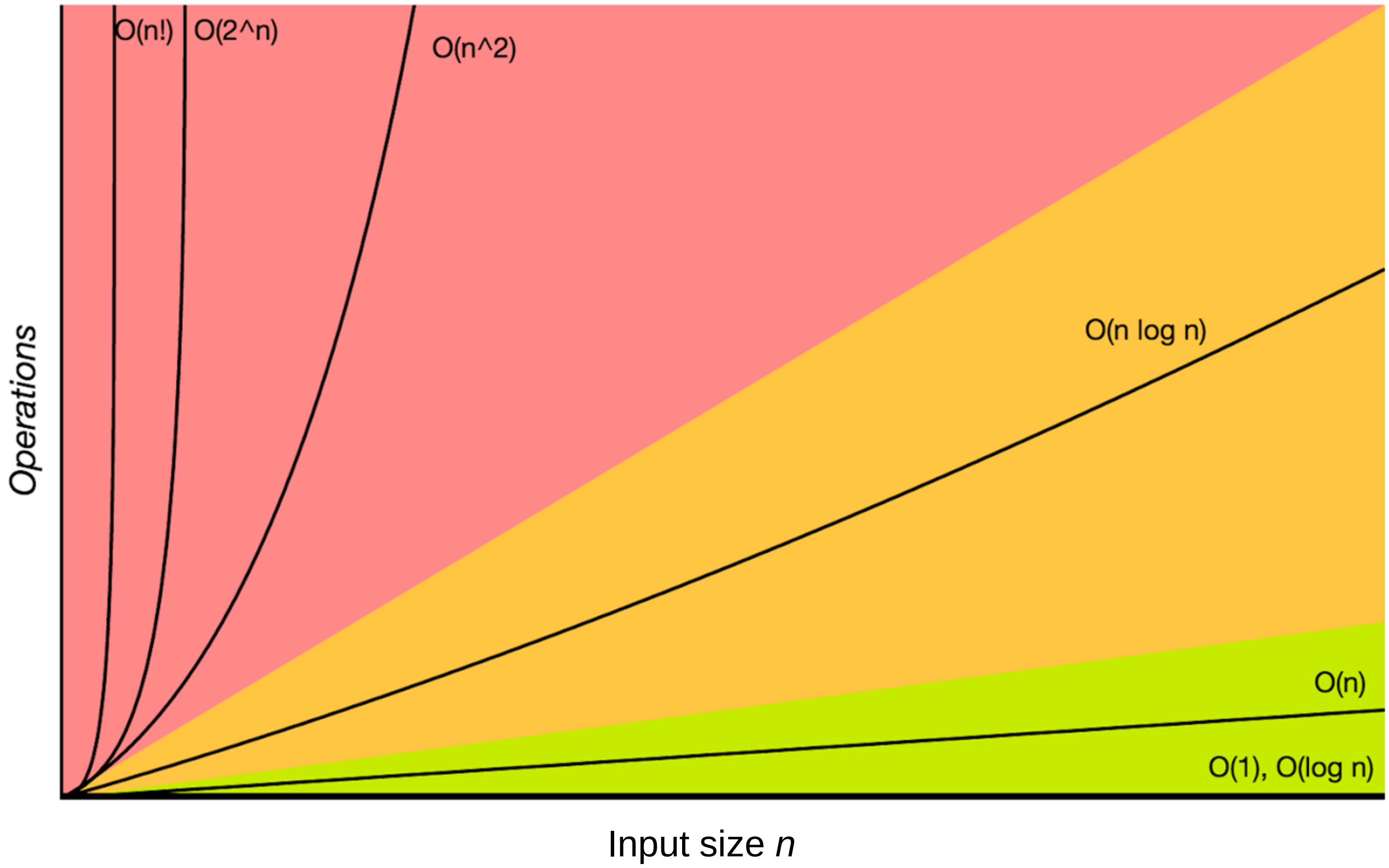
Lecture 8

Classifying algorithms by complexity
Use cases: *searching* and *sorting*

Algorithms: practical and impractical

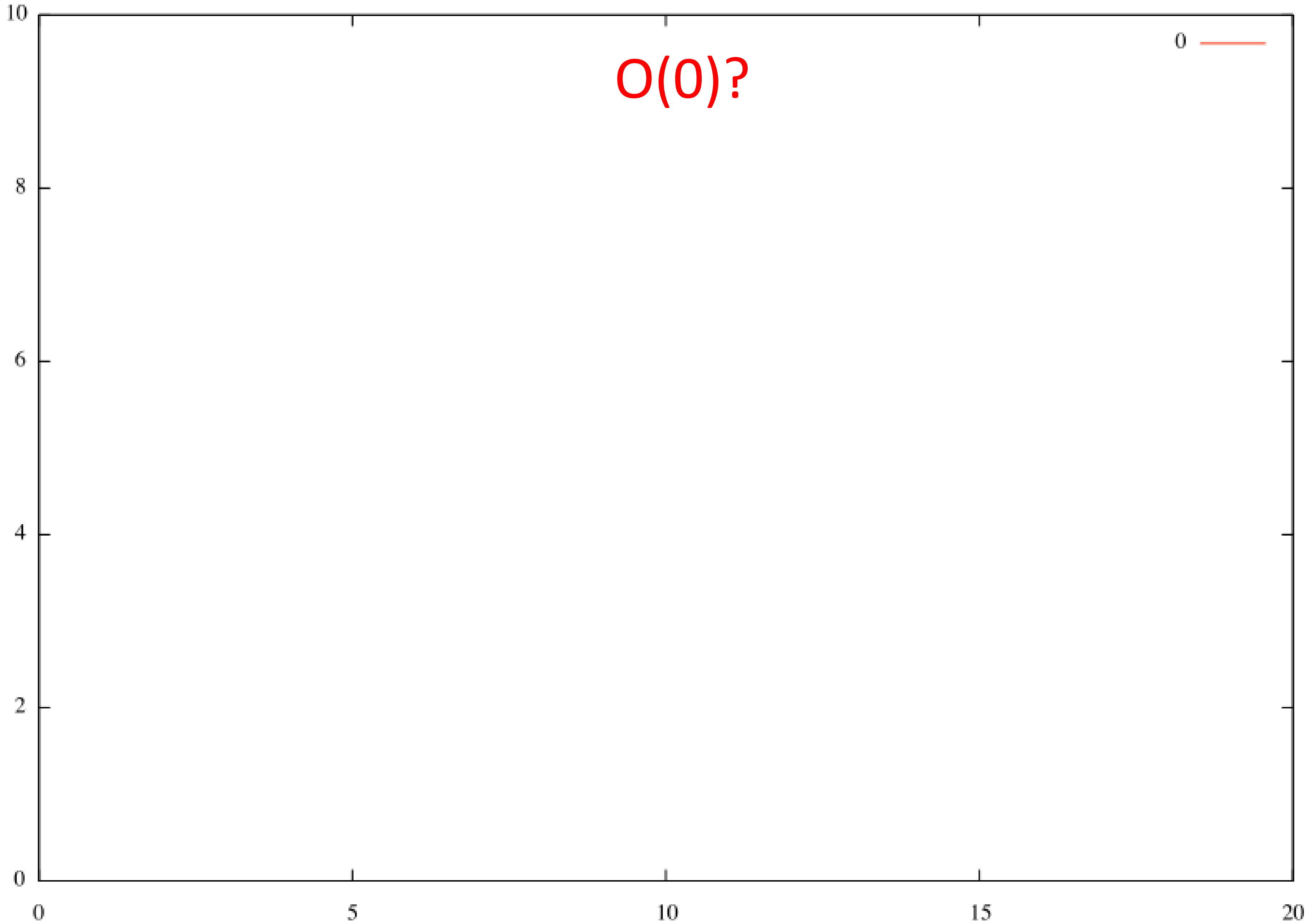


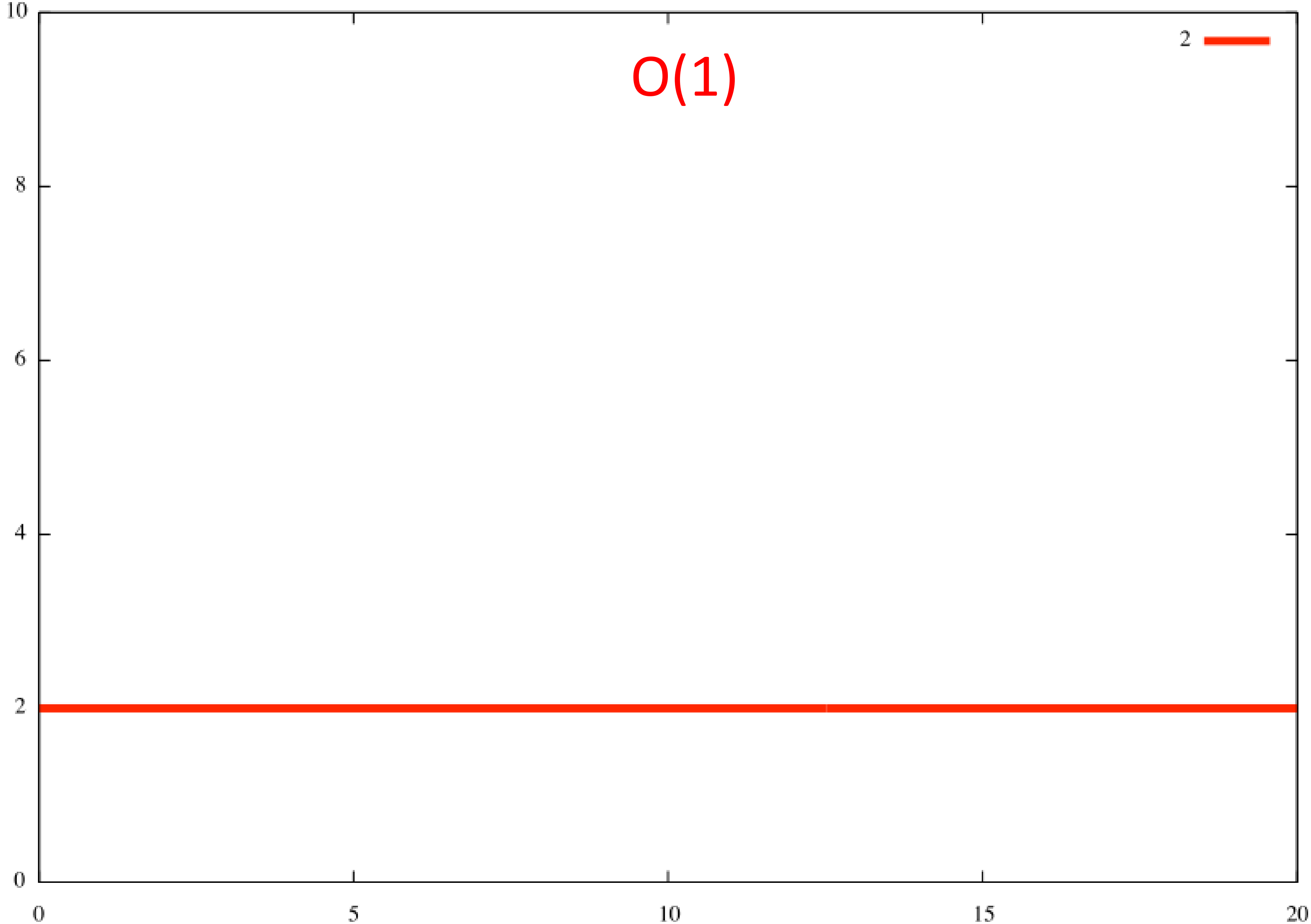
n bytes	log n	n	n^2	2^n
10 B	1	10	100	$\sim 1*10^3$
100 B	2	100	10000	$\sim 1*10^{30}$
1 KB	3	1,000	1000000	$\sim 1*10^{300}$
10 KB	4	10,000	100000000	$\sim 1*10^{3000}$
100 KB	5	100,000	10000000000	$\sim 1*10^{30,000}$
1 MB	6	1,000,000	1.00E+12	$\sim 1*10^{300,000}$
10 MB	7	10,000,000	1.00E+14	n/a
100 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

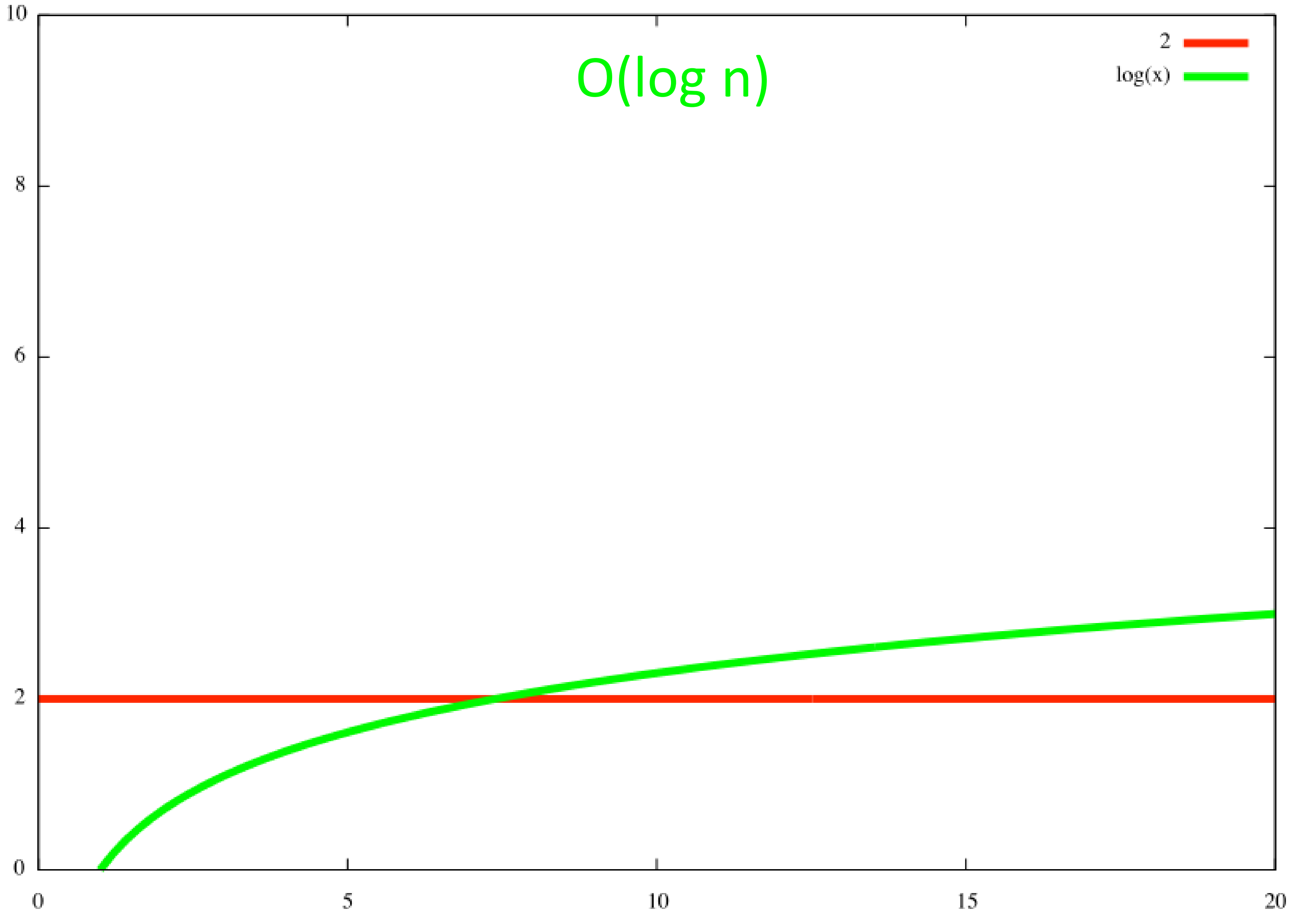
CPU with a clock speed of 2 gigahertz (GHz) can carry out two thousand million (**$2*10^9$**) cycles (operations) **per second**.

- Algorithm which runs in **$O(2^n)$** time will process **1 KB** of input in **$\sim 10^{22}$ years** (more than 7 millenia)
- 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

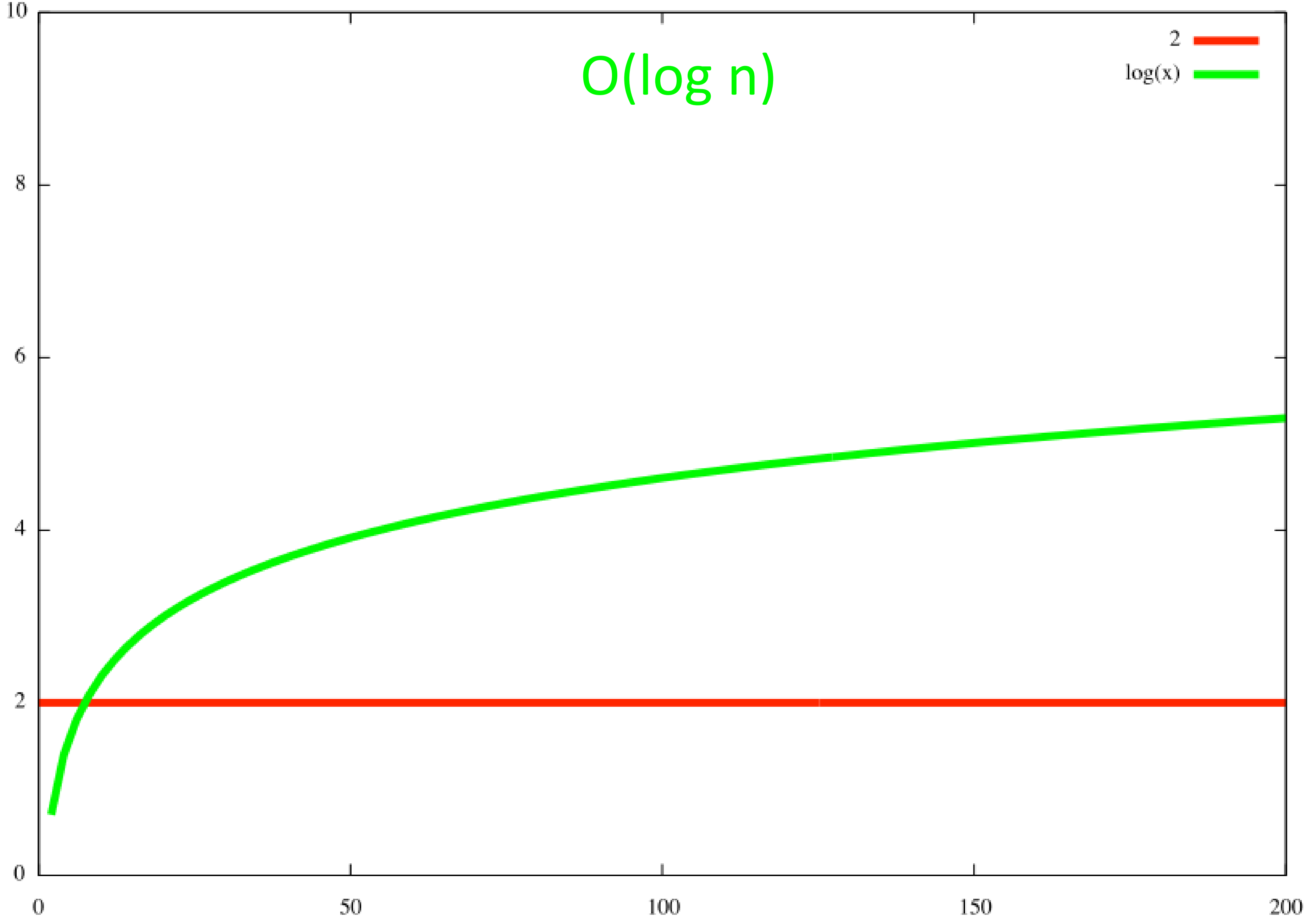
Classifying algorithms by the rate of growth

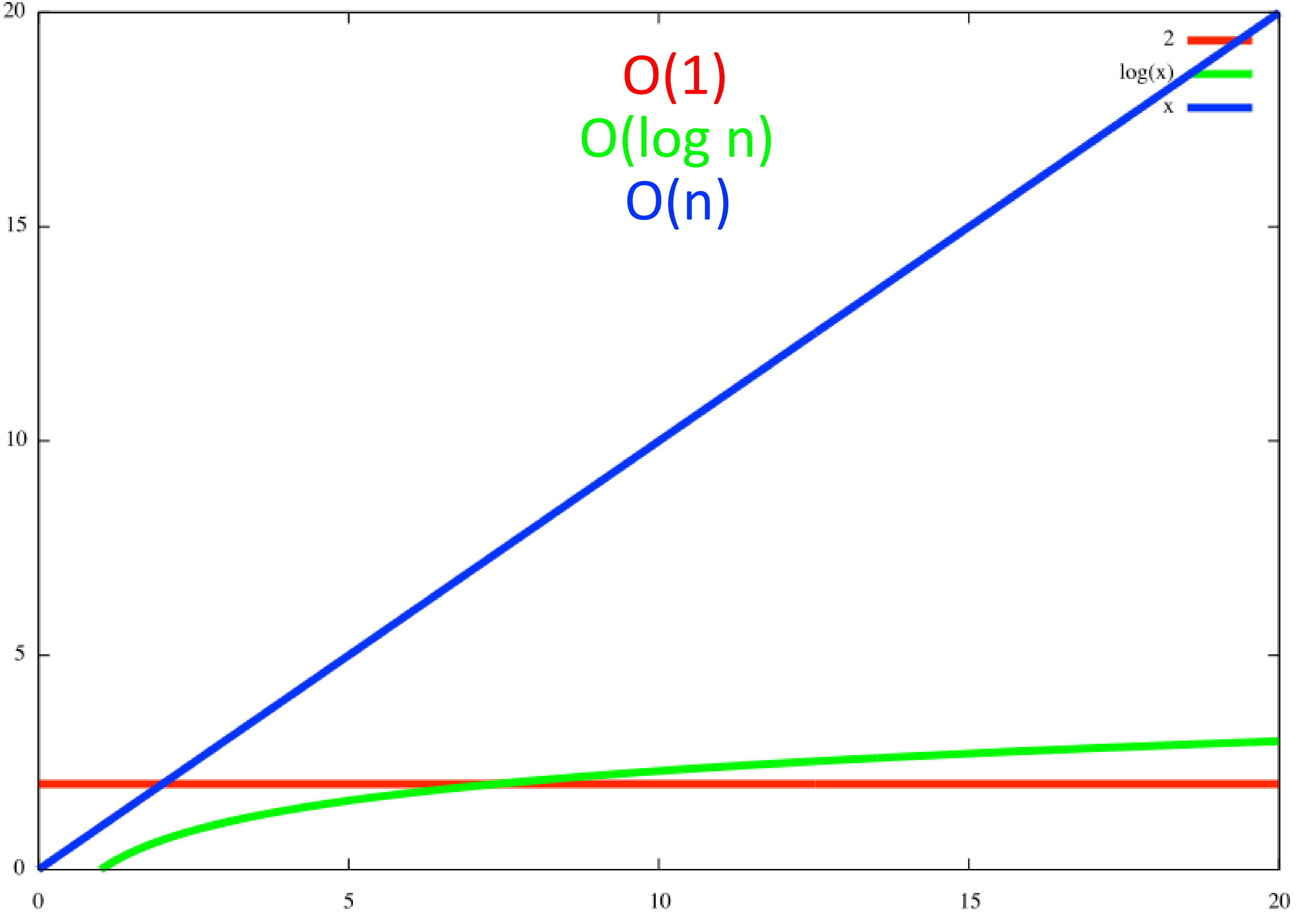

O(0)?
0
10
8
6
4
2
0
0
5
10
15
20


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

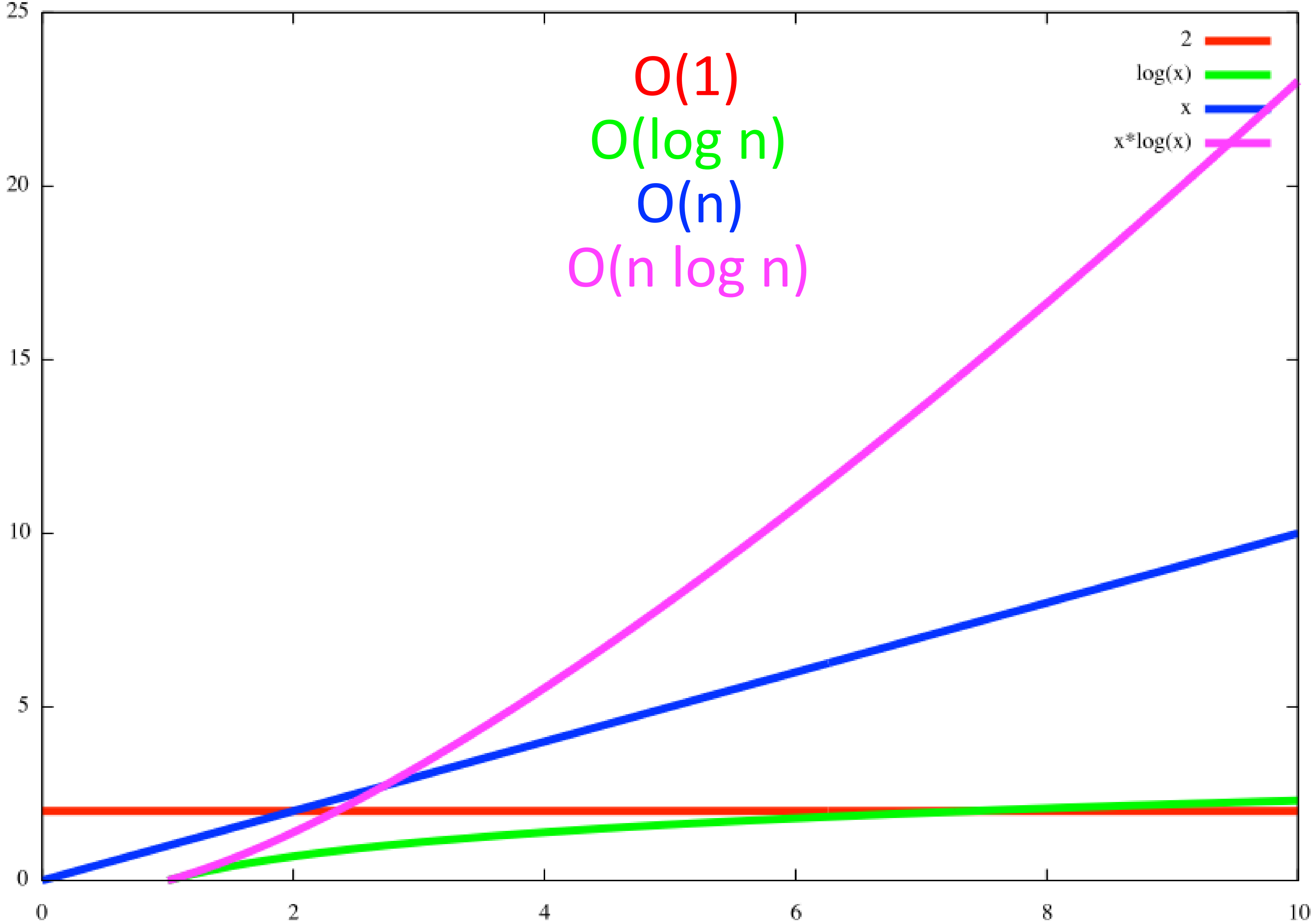


O(log n)
2
log(x)
10
8
6
4
2
0
0
5
10
15
20

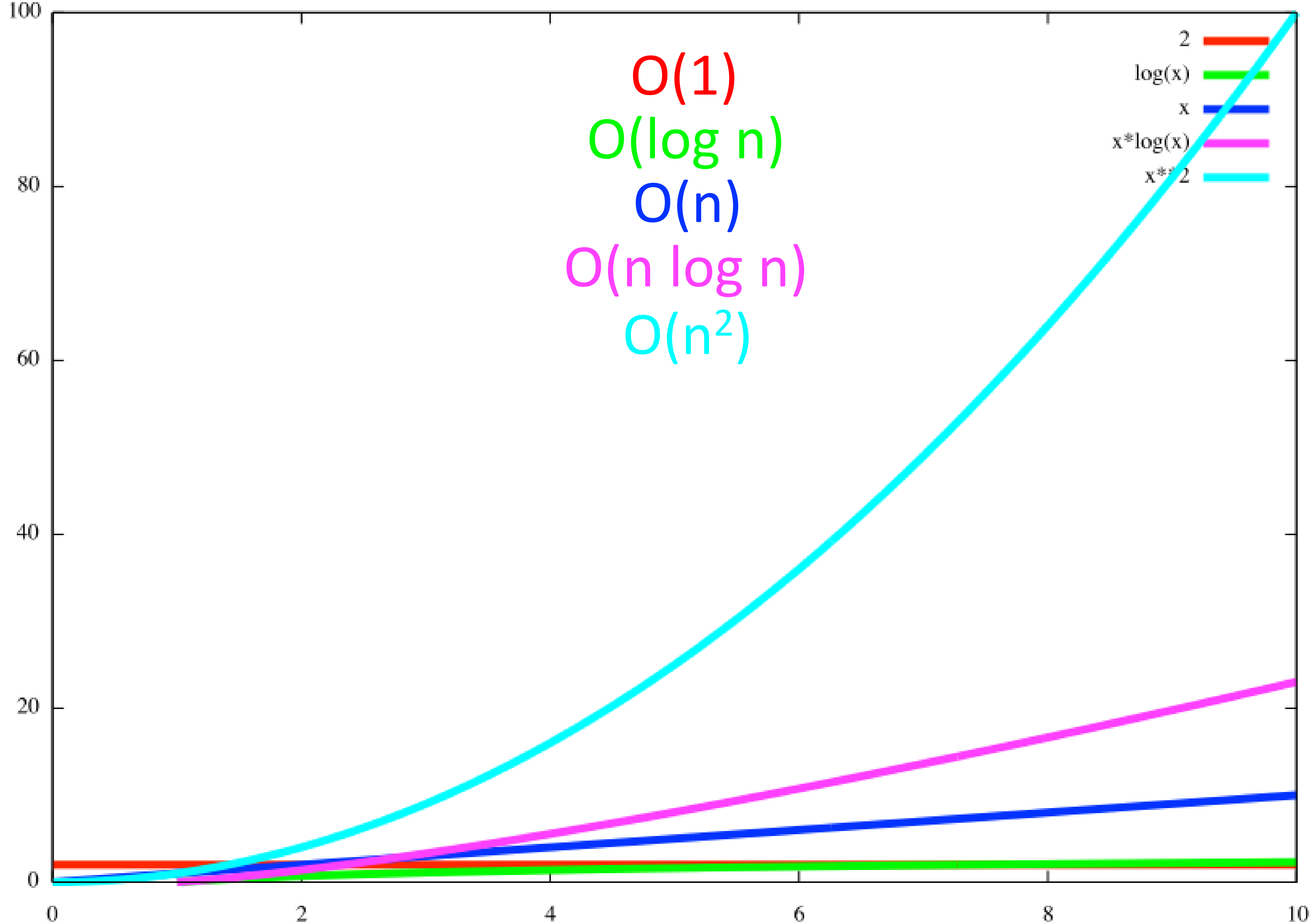

O(log n)
2
log(x)
10
8
6
4
2
0
0
50
100
150
200



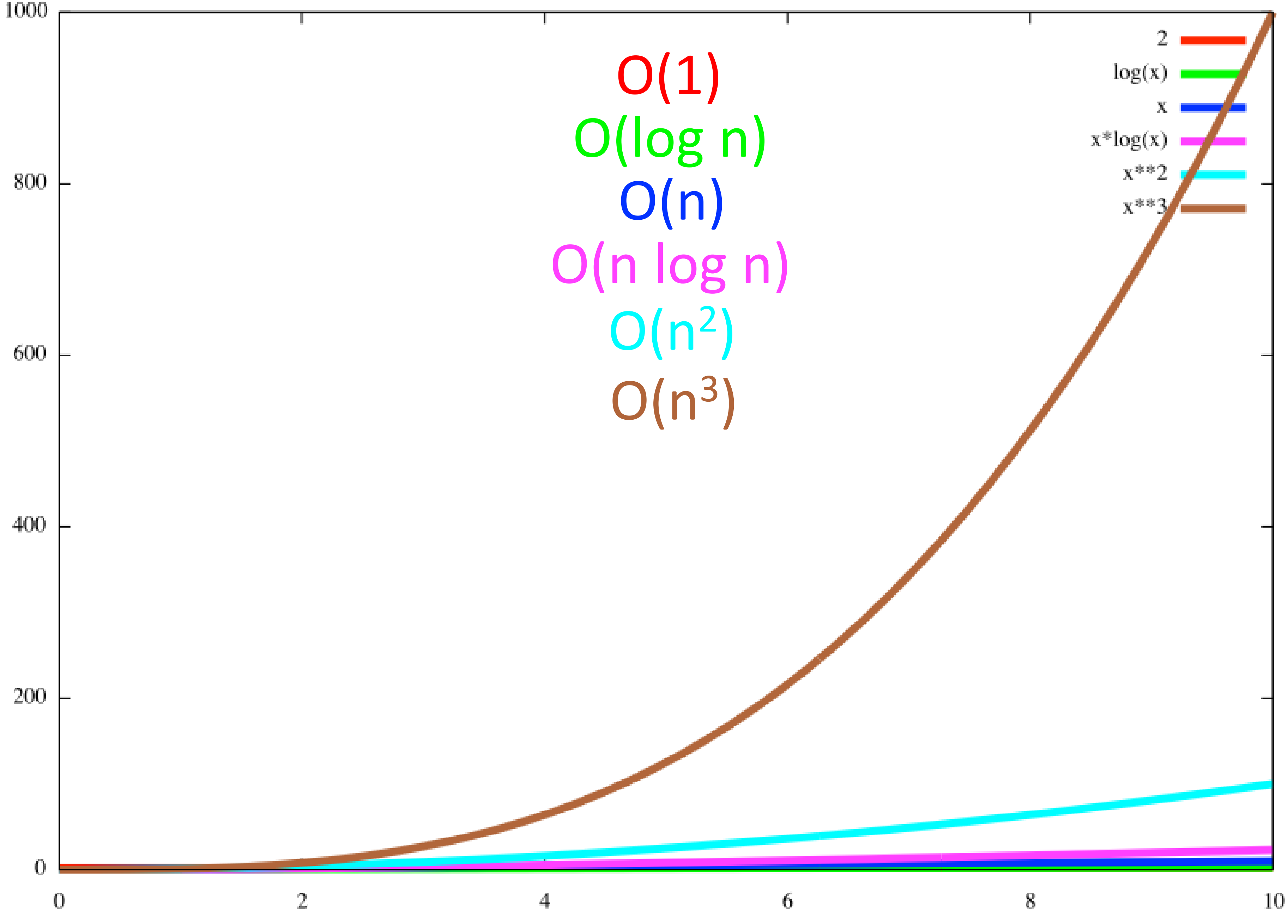
O(1)
O(log n)
O(n)
2
log(x)
x
0
5
10
15
20
0
5
10
15
20



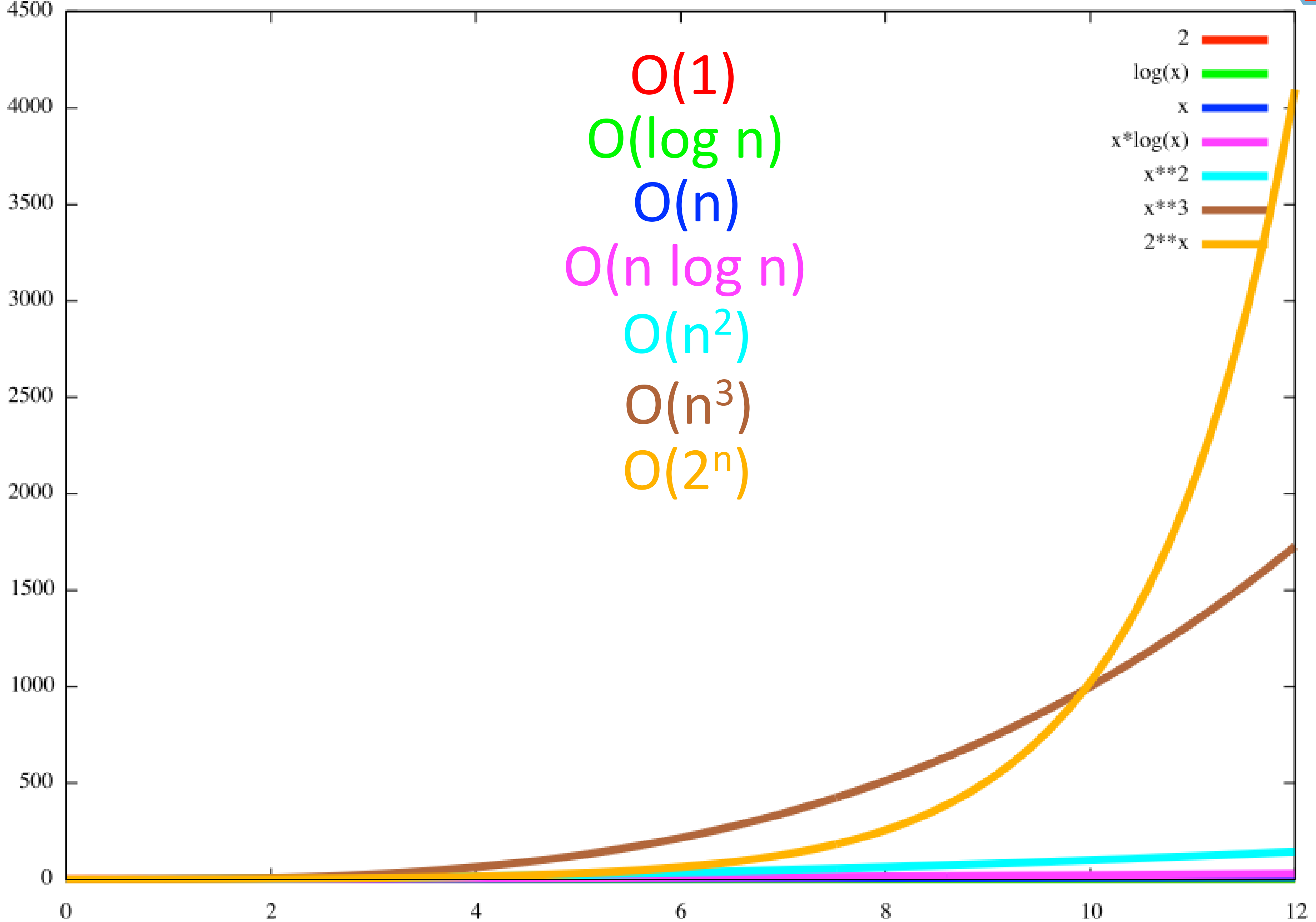
O(1)
O(log n)
O(n)
O(n log n)
2
log(x)
x
x*log(x)
25
20
15
10
5
0
0
2
4
6
8
10



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

Example:
Complexity of searching in an Array

Linear Search

Algorithm linearSearch (*A*, *n*, *target*)

```
i ← 0

while i < n:
    if target = A[i ]:
        return i
    i ← i + 1
return -1
```

Linear Search

Algorithm linearSearch (*A, n, target*)

```
i ← 0

while i < n:
    if target = A[i ]:
        return i
    i ← i + 1
return -1
```

$1 + 4n + 1 = 4n + 2 = \mathbf{O(n)}$

Searching Sorted Data

Binary Search

Searching in a Sorted Array

Example

$search(2) \longrightarrow$ -1

$search(3) \longrightarrow$ 0

$search(4) \longrightarrow$ -1

$search(20) \longrightarrow$ 3

$search(20) \longrightarrow$ 4

$search(90) \longrightarrow$ -1

3	5	8	20	20	50	60
0	1	2	3	4	5	6

Example: Searching for key 50

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

Example: Searching for key 50

binarySearch(A, 11, 50)

$low = 0$

$high = 10$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *high*

Example: Searching for key 50

binarySearch(A, 11, 50)

$low = 0$

$high = 10$

$mid = 0 + \text{floor}[(10 - 0)/2] = 5$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

binarySearch(A, 11, 50)

$low = 6$

$high = 10$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *high*

Example: Searching for key 50

binarySearch(A, 11, 50)

$low = 6$

$high = 10$

$mid = 6 + \text{floor}[(10 - 6)/2] = 8$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

binarySearch(A, 11, 50)
$low = 9$
$high = 10$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *high*

Example: Searching for key 50

binarySearch(A, 11, 50)

$low = 9$

$high = 10$

$mid = 9 + \text{floor}[(10 - 9)/2] = 9$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

$\text{binarySearch}(A, 11, 50) \rightarrow 9$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

Binary search

Algorithm binarySearch (*A, n, target*)

$low \leftarrow 0$

$high \leftarrow \text{n - 1}$

while *low* ≤ *high*:

$\quad mid \leftarrow \text{low} + \left\lfloor \dfrac{high - low}{2} \right\rfloor$

$\quad$ if $target = A[mid]$:

$\quad\quad$ return mid

$\quad$ else if $target < A[mid]$:

$\quad\quad high = mid - 1$

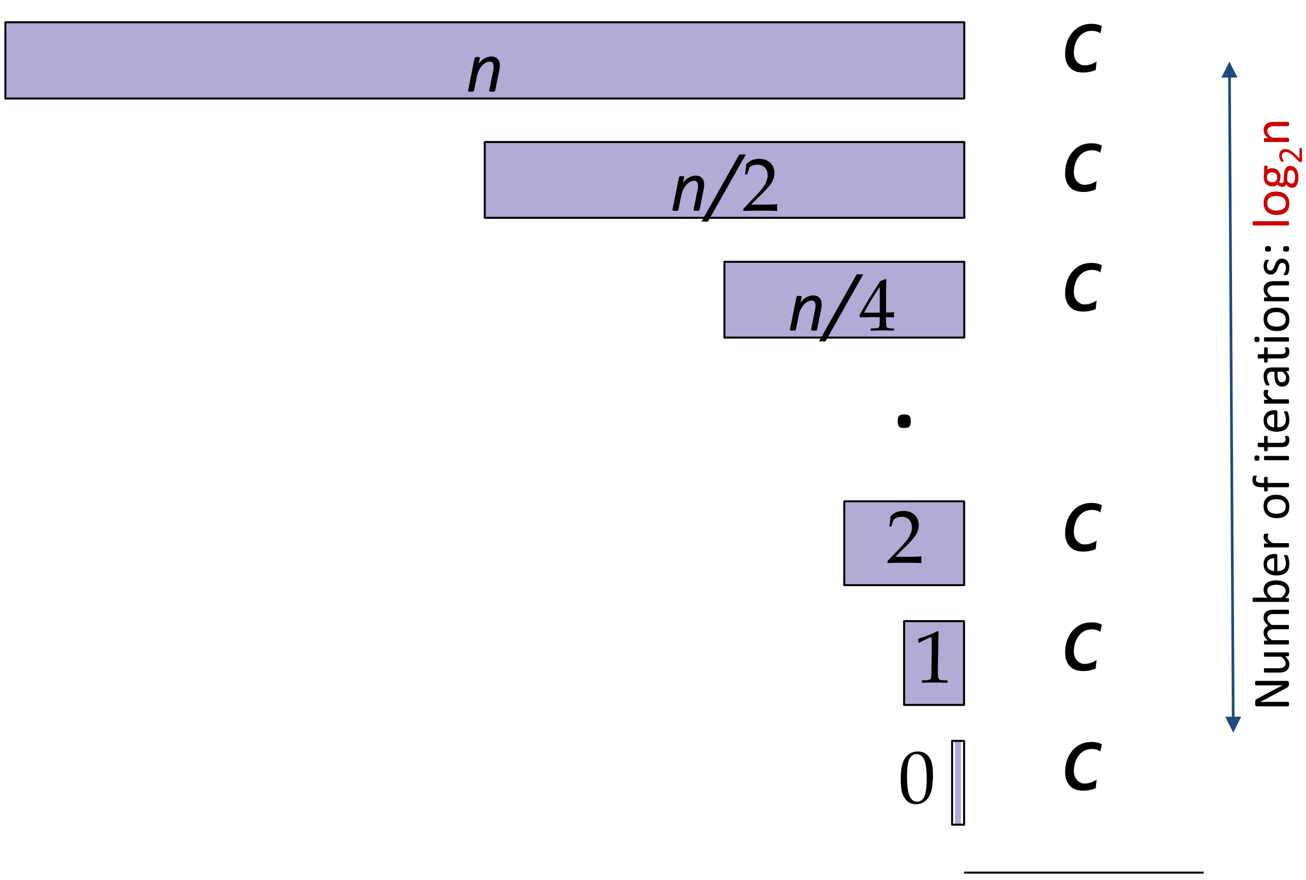
$\quad$ else:

$\quad\quad low = mid + 1$

return -1

Running time of Binary Search

Input size Work in each iteration

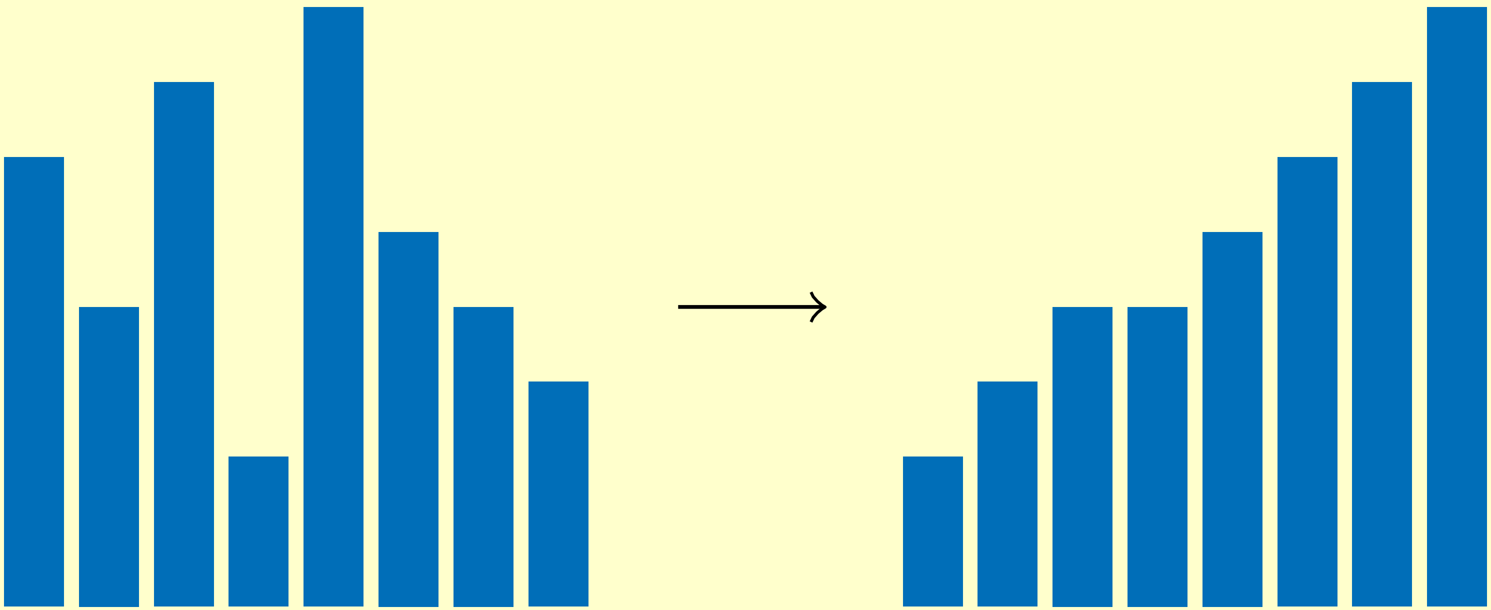


$$\text{Total: } \sum_{i=0}^{\log_2 n} c = O(\log n)$$

Linear search	Binary search
$O(n)$	$O(\log n)$

Searching in a sorted array is significantly faster

Sorting



Why Sorting?

- Sorting data is an important step of many efficient algorithms
- Sorted data allows for more efficient queries (binary search)

Bubble sort

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

Insertion sort

```
void insertionSort (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
```

Inefficient sorting algorithms:

- Selection sort: LINK
- Insertion sort: LINK
- Bubble sort: LINK

All these algorithms are **$O(n^2)$**

- Later in this course we will learn more efficient sorting that runs in time $O(n \log n)$