Complexity

Linear search (worst case complexity)

```
1  int search (int[] array, int x) {
2    int n = array.length;
3    int i = 0;
4
5    while (i < n) {              // iterate over the elements
6      if (array[i] == x) {
7        return i;                // found it!
8      } else {
9        i = i + 1;               // try the next one
10     }
11   }
12
13   return -1;                   // the value not found
14 }
```

Worst case: the value `x` is not in the array.

Number of steps: $2 + n \times (1 + 1 + 1) + 2 = 3n + 4$

(2nd and 3rd lines, then `n` -times 5th, 6th and 9th lines, and finally the 5th and 13th line)

Linear search (worst case complexity), recursively

```
1  int search (int[] array, int x) {
2    search_rec(array, 0, x);
3  }
4
5  int search_rec(int[] array, int i, int x) {
6    if (i == array.length)
7      return -1;                    // the value not found
8
9    if (array[i] == x)
10     return i;                     // found it!
11
12   int i_next = i + 1;             // try the next one
13   return search_rec(array, i_next, x);
14 }
```

Worst case: the value `x` is not in the array.

Number of steps: $1 + n \times (1 + 1 + 1 + 1) + 3 = 4n + 4$

(2nd line, then `n` -times 6th, 9th, 12th and 13th lines, and finally 6th and 7th line)

What is the difference between the two?

From the theoretical perspective we are more interested in how the number of steps *grows with respect to the input size*, rather than in the actual number of steps. This is because the actual speed depends on

- the hardware on which it runs,
- programming language used (or its compiler),
- how well is the implementation optimised, ...

Performance of Algorithms

A number of timed searches were performed on a sorted list of 10,000 numbers. When searching for a number at the start, at the end, and that was not in the list respectively, the timings were in the following ranges:

- Linear search:
  - Start: 2 μsecs
  - End: 391 μsecs
  - Not in: 356 μsecs
- Binary search:
  - Start: 6 μsecs
  - End: 5 μsecs
  - Not in: 5 μsecs

Performance of Algorithms

So: Binary search seems to be faster than linear search

- unless searching for the first item in the list
- on the machine that these timings were run on
- when run on sorted lists of numbers
- when the list is of length 10,000
- ... and possibly with other restrictions

We need a better way to be able to think about and compare algorithm performance.

Time and Space Performance

There are 2 *dimensions* of performance we might be interested in:

- Time: How long it takes to run the algorithm:
  - Measuring this in normal time units makes us dependent on the machine we run it on.
  - Instead measure it in numbers of steps: e.g. the number of additions or multiplications, the number of comparison operations, the number of memory accesses etc.
- Space: How much memory it requires:
  - Different algorithms to accomplish the same result might use different amounts of memory. For example:
    - One takes 1,000,000 steps and require only 2 integer variables
    - Another take 20,000 steps but requires a list of length 1,000

To choose between algorithms, we need to understand both its time and space performance.

Complexity

Even if we know an algorithm's time performance (in units of steps) and space performance (in units of words of memory), we still do not yet have a way of capturing how that performance changes with different sizes of problems.

Solution: parameterize the performance by the size of the input:

- This algorithm takes N steps on an input of size N
- This algorithm takes $2N^2 + N$ steps on an input of size N
- This algorithm uses $3N$ words of memory on an input of size N

Average and Worst Case Complexity

Linear search took different times depending on whether the item was at the start of the list or at the end even though the size of the list didn't change:

- *Average case time complexity*: Consider every possible case for an input of size n and calculate the average
  - Linear search on n elements requires 1 comparison if the item is first in the list.
  - It requires 2 comparisons if the item is second in the list...
  - It requires n comparisons if the item is last in the list.
  - Average time complexity: $\frac{1+2+\cdots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$ comparisons
- *Worst case time complexity*: Choose the case that will take the largest number of steps.
  - Linear search for the last item in the list: n comparisons

Worst Case Complexity of Binary Search

How large a list, n can we search with c comparisons?

$$
\begin{aligned}
1: \quad & 1 & = \; & 1 = 2^1 - 1 \\
2: \; & 1 + 1 + 1 & = \; & 3 = 2^2 - 1 \\
3: \; & 3 + 1 + 3 & = \; & 7 = 2^3 - 1 \\
4: \; & 7 + 1 + 7 & = \; & 15 = 2^4 - 1
\end{aligned}
$$

In general: c comparisons lets us search a list of length $2^c - 1$

We want to know the inverse: how many comparisons do we need to search a list of length n?

If we ignore the "-1", which only has a small relative effect, the worst time complexity of binary search is $log_2(n)$

Average Case Complexity of Binary Search

What about the average case? Take a list of length n:

- Only 1 case where we find the target item in 1 comparison
- 2 cases where we find the item in 2 comparisons
- 4 cases where we find the item in 3 comparisons
- $\vdots$
- $n/2$ cases where we find the item in $log_2(n)$ comparisons

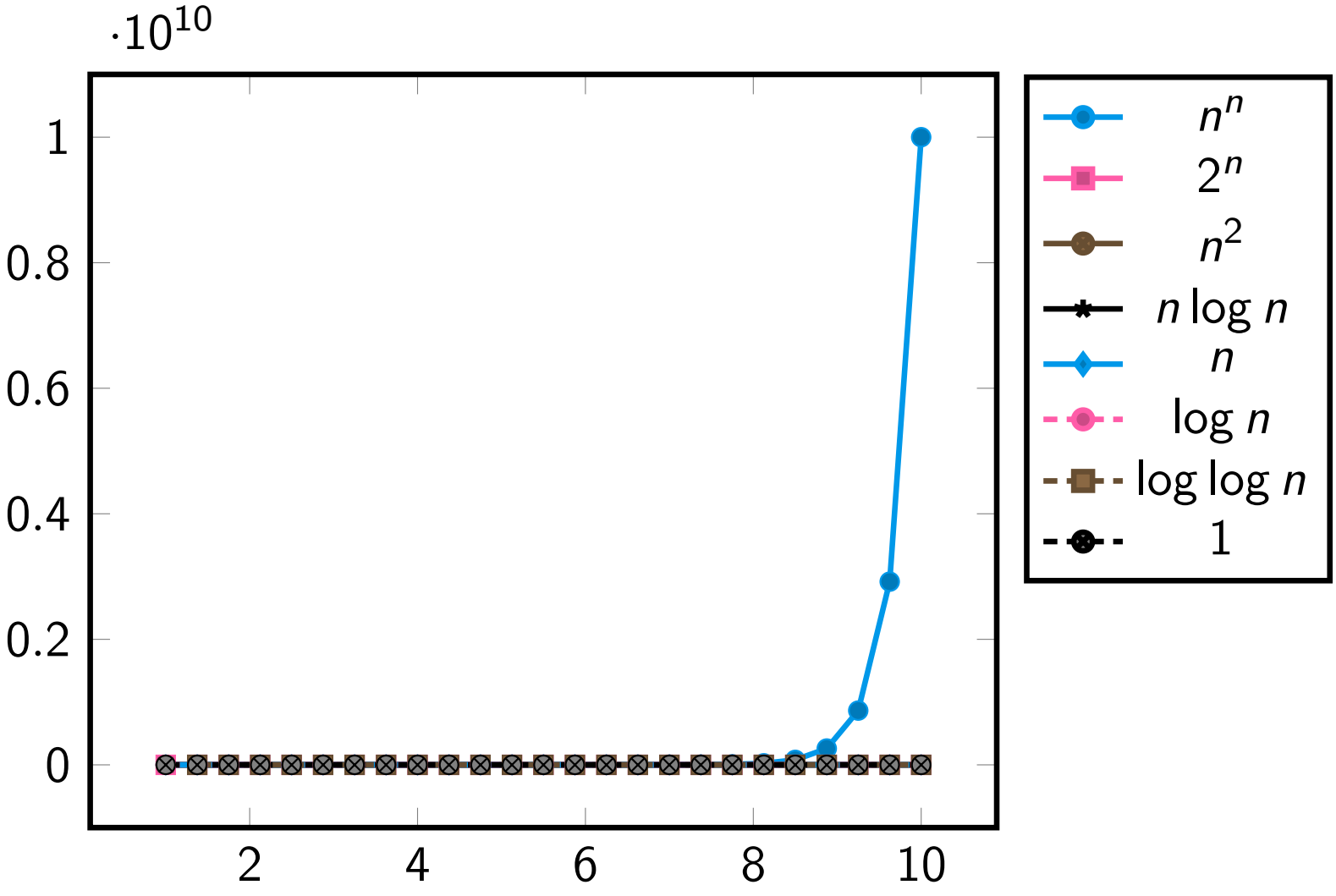
Thus n cases in total where:

- Half have the worst case complexity of $log_2(n)$
- One quarter have complexity $log_2(n) - 1$
- One eighth have complexity $log_2(n) - 2$
- $\vdots$

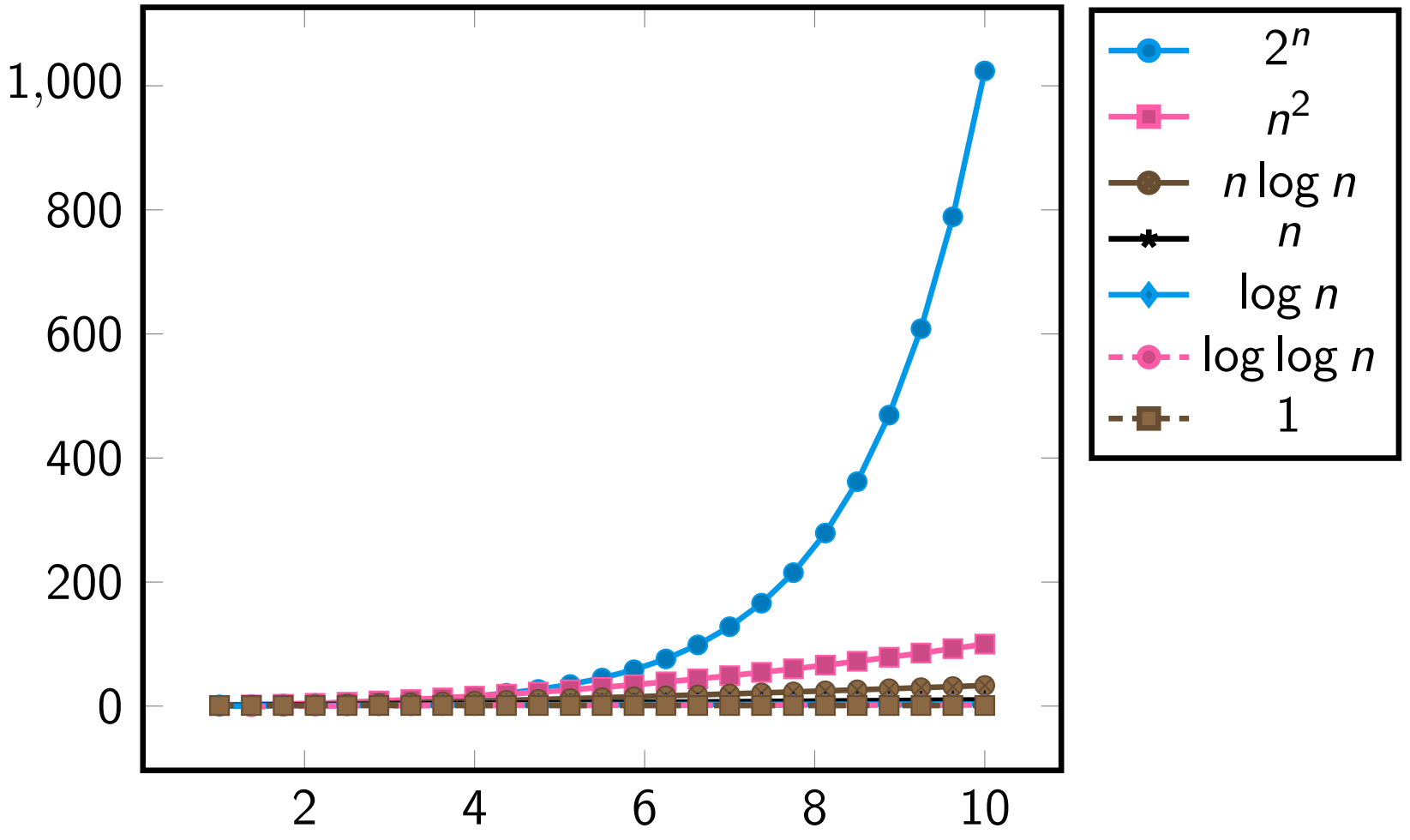
Average case is only slightly less than the worst case:

Approximate it with the worst case.

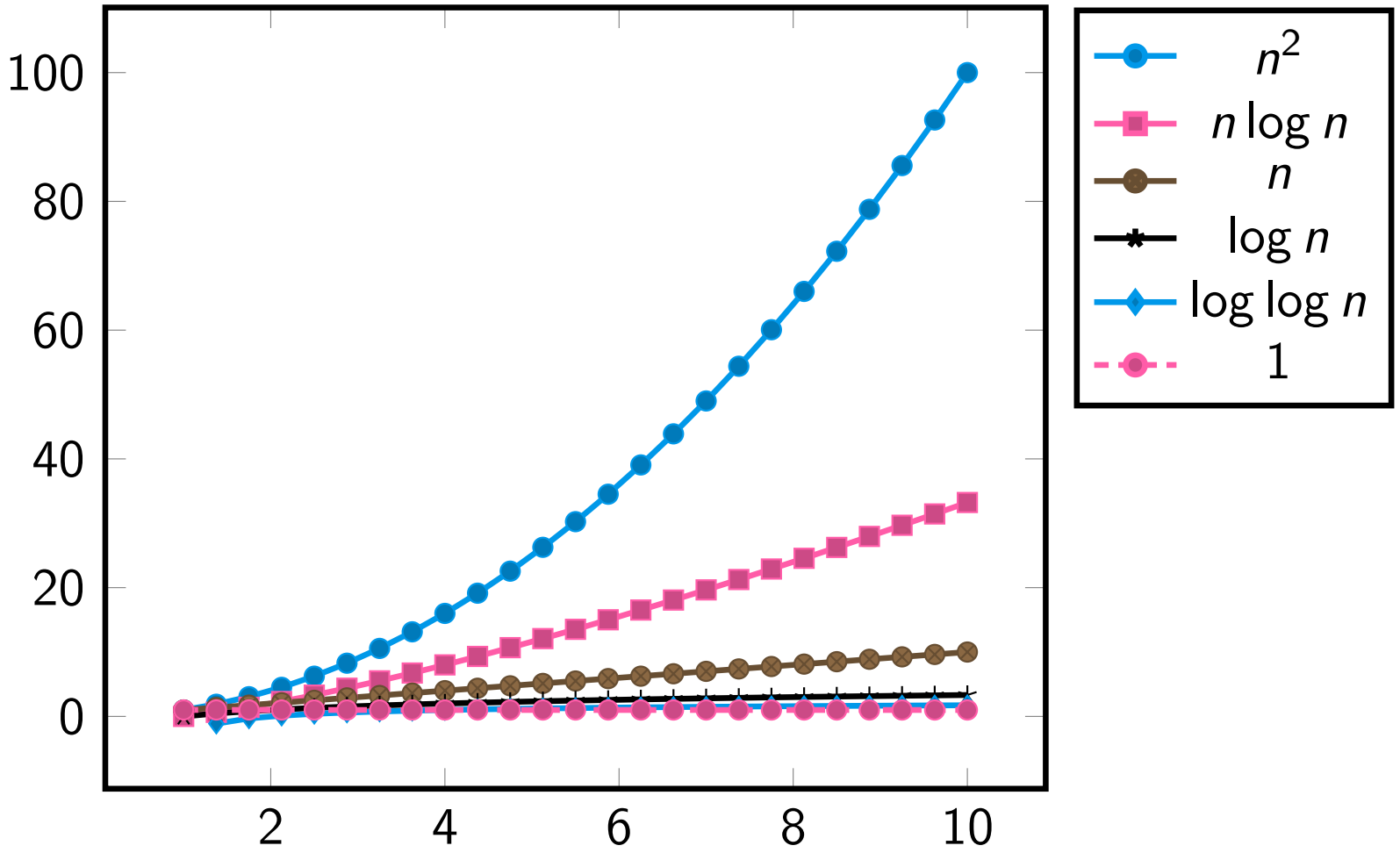
Comparing Functions



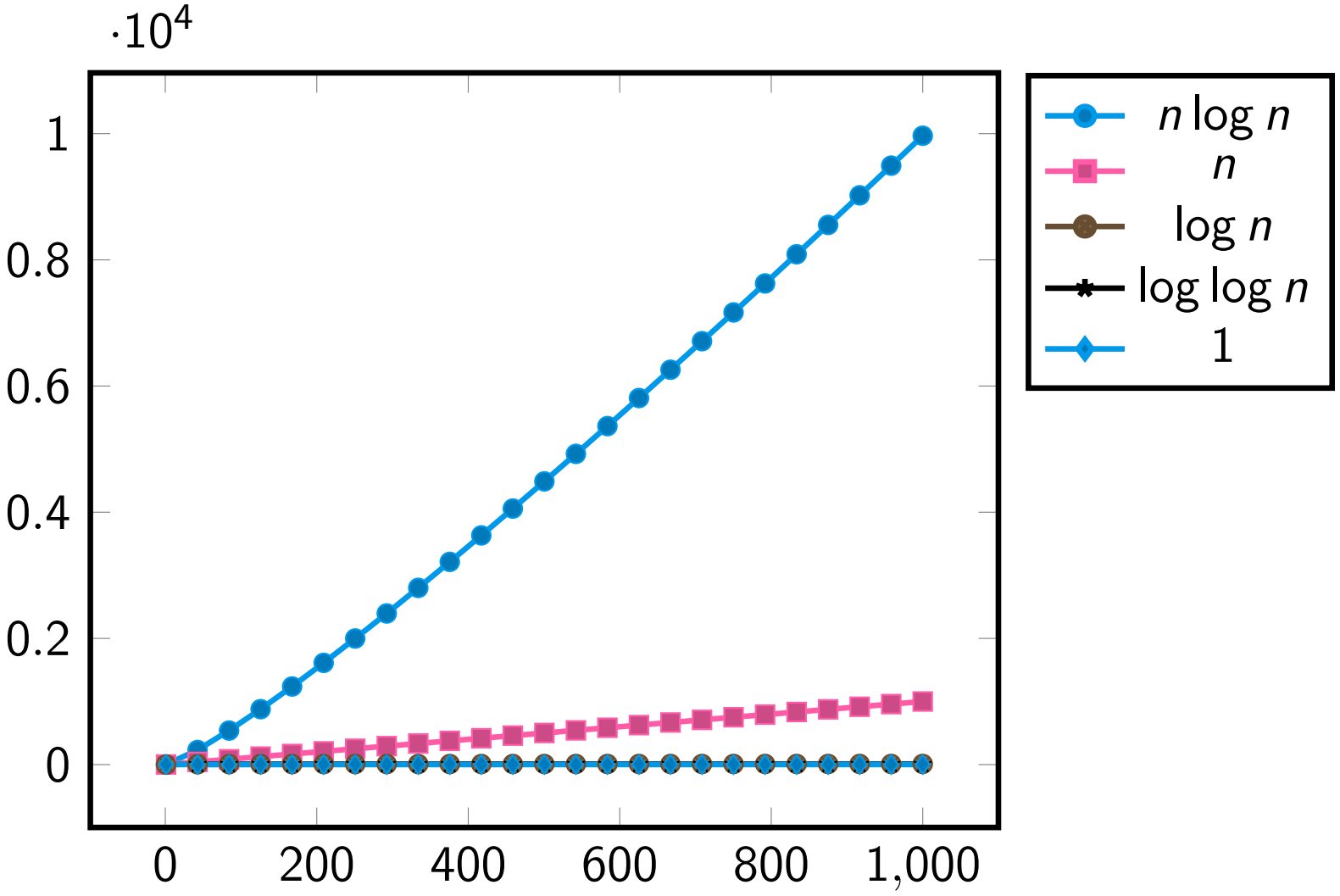
Comparing Functions



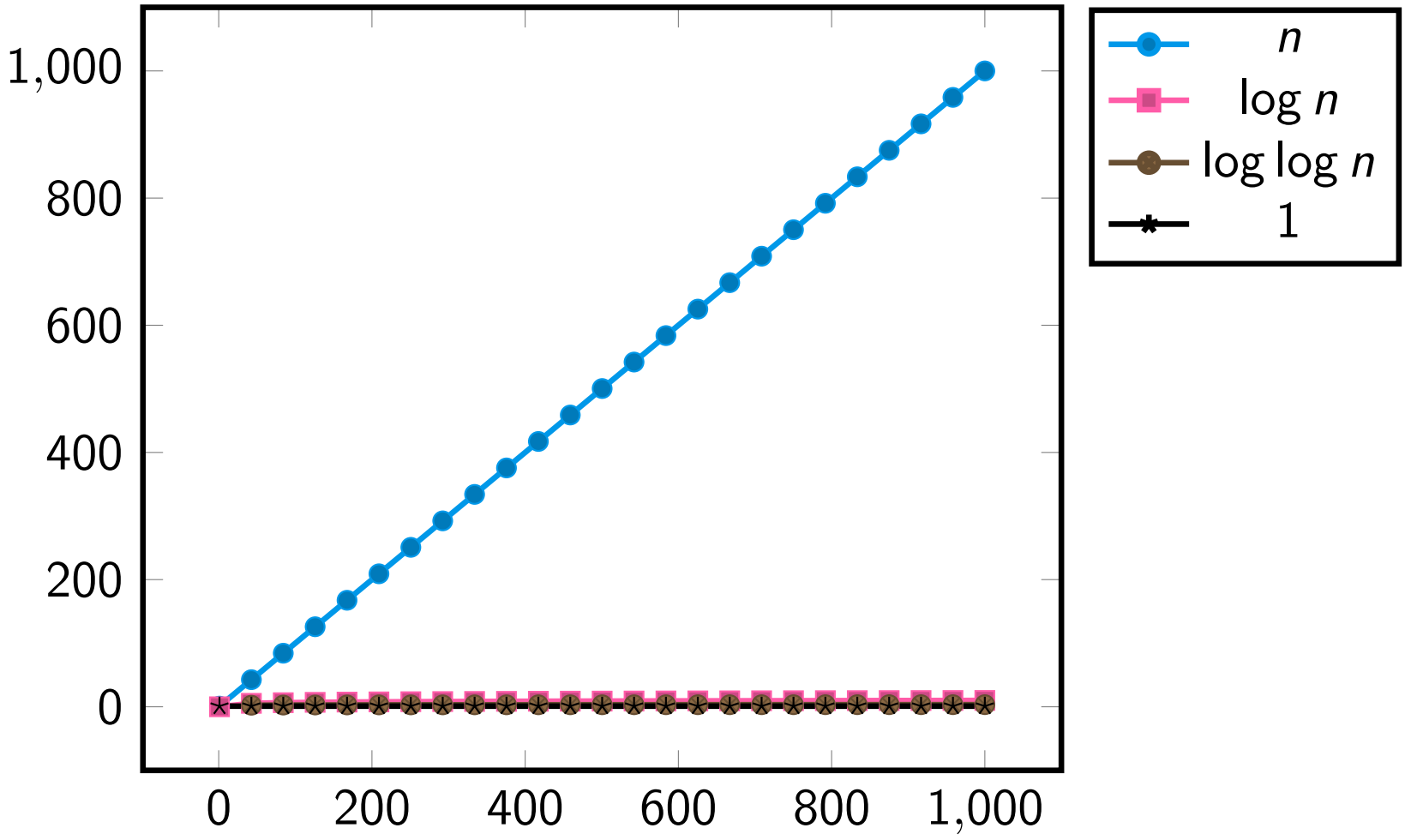
Comparing Functions



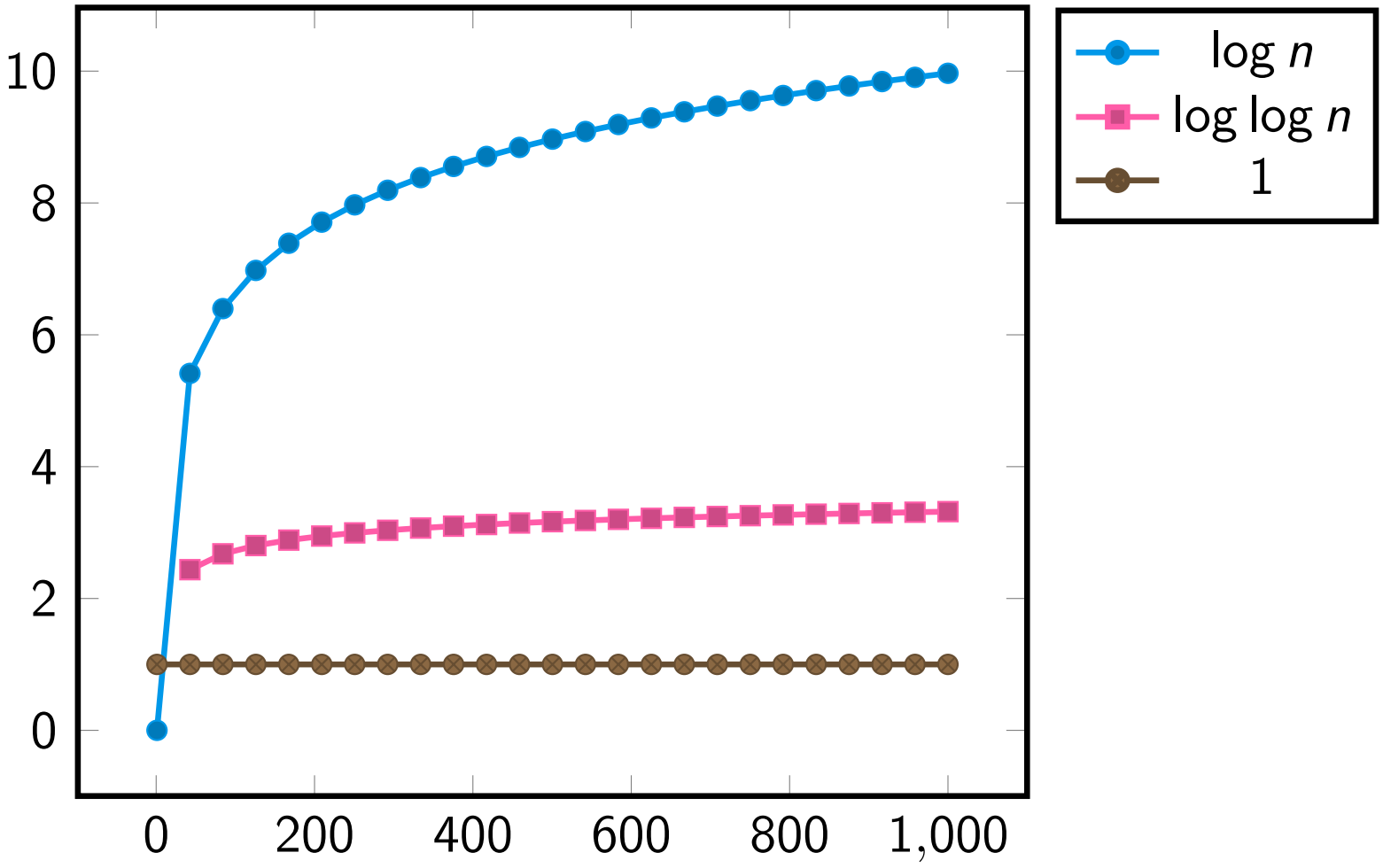
Comparing Functions



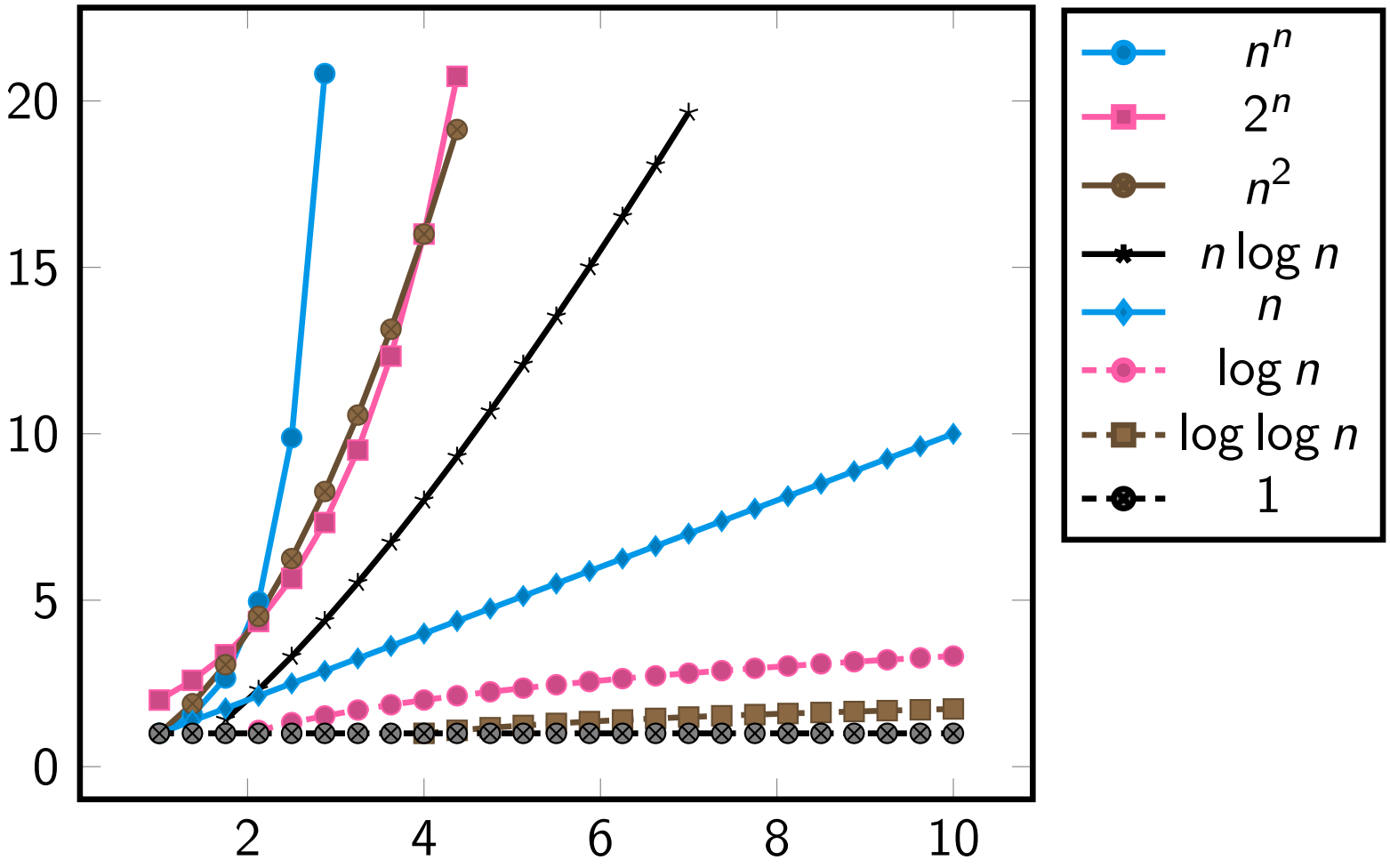
Comparing Functions



Comparing Functions



Comparing Functions



Big O notation

The precise number of steps is often too detailed to get a clear understanding of the performance of an algorithm:

- What if an algorithm does a comparison and a multiplication on every element of a list:
  - Complexity is n steps, where a step is a comparison **AND** a multiplication
  - Complexity is $2n$ steps, where a step is a comparison **OR** a multiplication

The difference between an algorithm that has time complexity n and one that has $2n$ is small in relation to the difference between two algorithms that have respective complexities of n and n^2.

Similarly in an algorithm that has complexity $n^2 + n$, the n part only makes a small contribution.

Solution: simplify to headline complexity

Big O notation

Describe complexity by the most significant overview feature:

$$f(n) = O(g(n)) \iff |f(n)| \leq |Cg(n)|$$

for positive constants C, n_0 where $n > n_0$

Idea: f does not grow at a faster rate than g as n increases. It might grow at the same rate or it might grow at a slower rate.

Examples (restrict ourselves to $n \geq 4$):

- $2n = O(n)$?
- $2n + 100 = O(n)$?
- $n = O(1)$?
- $3n^2 + n = O(n^2)$?
- $3n^2 + n = O(n^3)$?
- $3n^2 + n = O(n)$?

Big O notation

Examples (restrict ourselves to $n \geq 4$):

- $2n = O(n)$?
  - *TRUE*: choose C to be 3
- $2n + 100 = O(n)$?
  - *TRUE*: choose C to be 1000
- $n = O(1)$?
  - *FALSE*: no value of C is large enough so that $n \leq C$
- $3n^2 + n = O(n^2)$?
  - *TRUE*: $3n^2 + n \leq 3n^2 + n^2 = 4n^2$ choose C to be 5
- $3n^2 + n = O(n^3)$?
  - *TRUE*: $3n^2 + n \leq 4n^2 \leq 4n^3$ choose C to be 5
- $3n^2 + n = O(n)$?
  - *FALSE*: no value of C is large enough so that $3n^2 + n \leq Cn$

More Big O examples

For all values of $n \geq 4$:

$$1 \leq \log\log n \leq \log n \leq n \leq n\log n \leq n^2 \leq 2^n \leq n^n$$

Therefore

$$\begin{aligned} 1 &= O(\log\log n) \\ \log\log n &= O(\log n) \\ \log n &= O(n) \\ n &= O(n\log n) \\ n\log n &= O(n^2) \\ n^2 &= O(2^n) \\ 2^n &= O(n^n) \end{aligned}$$

Thus, for example:

$$3n^n+42^n+100n^2+454n\log n+24n+12\log n+52\log\log n+43 = O(n^n)$$

Be careful with Big O

Big O is a *notation*, not a function. Thus $O(f(n))$ is not a function, even if it looks like one:

- "$3n^2 + 4 = O(n^2)$" is just a shorthand way of writing "$|3n^2 + 4| \leq |Cn^2|$ for some constant C"
- "$O(n^2) = 3n^2 + 4$" does not have any meaning
  - If it did, we could do nasty things like:

$$3 = O(1) = 2 \text{ hence } 3 = 2 \textbf{ WRONG!!}$$

Big O notation can be made mathematically precise by defining it to be the *class* of functions with complexity $O(f(n))$. Therefore we can say that the complexity of an algorithm is "*in*" $O(f(n))$ or, shortening it, simply say that, for an algorithm X, we have that $X \in O(f(n))$

Linear search (average case complexity)

```java
int search (int[] array, int x) {
  int n = array.length;
  int i = 0;

  while (i < n) {
    if (array[i] == x) {
      return i;
    } else {
      i++;
    }
  }

  return -1; // the value not found
}
```

Average case: the value `x` is on the position $\frac{n}{2}$ (We assume that `x` appears once in the array)

Number of steps: $2 + \frac{n}{2} \times 3 = \frac{3}{2}n + 2 \quad \Longrightarrow \quad$ it is in $O(n)$

(one iteration of the while loop is 3 steps, no matter if we found `x` or not)

Previously we computed that the **worst case** complexity of linear search is $O(n)$. This happens if the value is not in the array and we need to search through the whole array.

Next, we consider a situation when the value `x` is in the array. (And for simplicity we assume that it is there only once). How many steps does it take **on average** to find `x`?

Because `x` can be on any position, it is on average in the middle of the array. This means that we find it on position $\frac{n}{2}$ and so the while-loop evaluates $\frac{n}{2}$-many times.

Binary Search (worst case and average case)

Searching `x` in a **sorted** array `arr`:

1. Compare `x` and `arr[arr.length div 2]`.
2. If `x` is bigger, recursively search `arr` on positions `(arr.length div 2) + 1`, ..., `arr.length - 1`.
3. Otherwise, recursively search `arr` on positions `0`, ..., `arr.length div 2`
4. We continue like this until we are left with only one element in the array. Then, return whether this element equals `x`.

The length of the array we search through reduces by one half in every step and we continue until the length is 1.

For simplicity assume that the length of `arr` is $n = 2^k$

$\Longrightarrow$ the number of steps is $O(k) = O(\log_2 n)$.