Lecture 6: Computational complexity, dynamic programming

- Time complexity: Big O notation
- Recursive functions
- Dynamic programming: Levenshtein distance

Midterm exam

- April 24, 10:15-11:00 (first half of the lecture), AND-3-02/06
- Pen-and-paper, multiple-choice and short text answers (no writing code)
- **Not** allowed: computers, documentation, slides, cheat sheet, any other material or devices
- More information on OLAT ("Exercise & Exam Info")

Learning objectives

By the end of this lecture, you should:

- Understand what computational complexity is and why it is important
- Be able to determine and reduce the time complexity of simple algorithms
- Know the time complexity of some commonly used operations with `list`s, `set`s, and `dict`s
- Understand how recursion works and be able to write recursive functions
- Know what dynamic programming is and why it is useful
- Understand the dynamic programming algorithm for calculating the Levenshtein distance

Imports

In [1]:

```
import random
import string
import timeit

import utils
```

How can we measure the efficiency of a program?

What resources does a program need?

- Time (seconds)
- Memory (bytes)
- Network data (megabits)
- Power (kilowatt-hours)
- ...

How to measure usage of these resources?

How to measure usage of these resources?

- **Benchmarking:** Measure how many resources the program uses in absolute units
    - Requires running the program (many times, maybe under different conditions)
    - Depends on input data, hardware, and other factors

How to measure usage of these resources?

- **Benchmarking:** Measure how many resources the program uses in absolute units
  - Requires running the program (many times, maybe under different conditions)
  - Depends on input data, hardware, and other factors

- **Computational complexity:** Determine how quickly runtime increases with increasing input length
  - Based on inherent characteristics of the program
  - Requires theoretical analysis of the code
  - Independent of hardware (can be done with pen and paper)

Two types of computational complexity

- **Time complexity:** How complex is our program in terms of the **time** it takes to run?
- **Space complexity:** How complex is our program in terms of the **memory** it takes to run?

Computational complexity tells us how **scalable** our algorithms are (e.g., with increasing corpus size, document length, vocabulary size, etc.)

Time complexity

Given an algorithm, how quickly does the **number of operations** grow when we increase the **input length**?

Time complexity

Given an algorithm, how quickly does the **number of operations** grow when we increase the **input length**?

1. For each operation, count how many times it is called
2. Sum up the counts
3. Keep only highest-order terms, ignore constant factors

In [2]:

```python
def minimum(numbers):
    min_number = float("inf")     # Called 1 time
    for number in numbers:
        if number < min_number:   # Called n times
            min_number = number   # Called n times
    return min_number
```

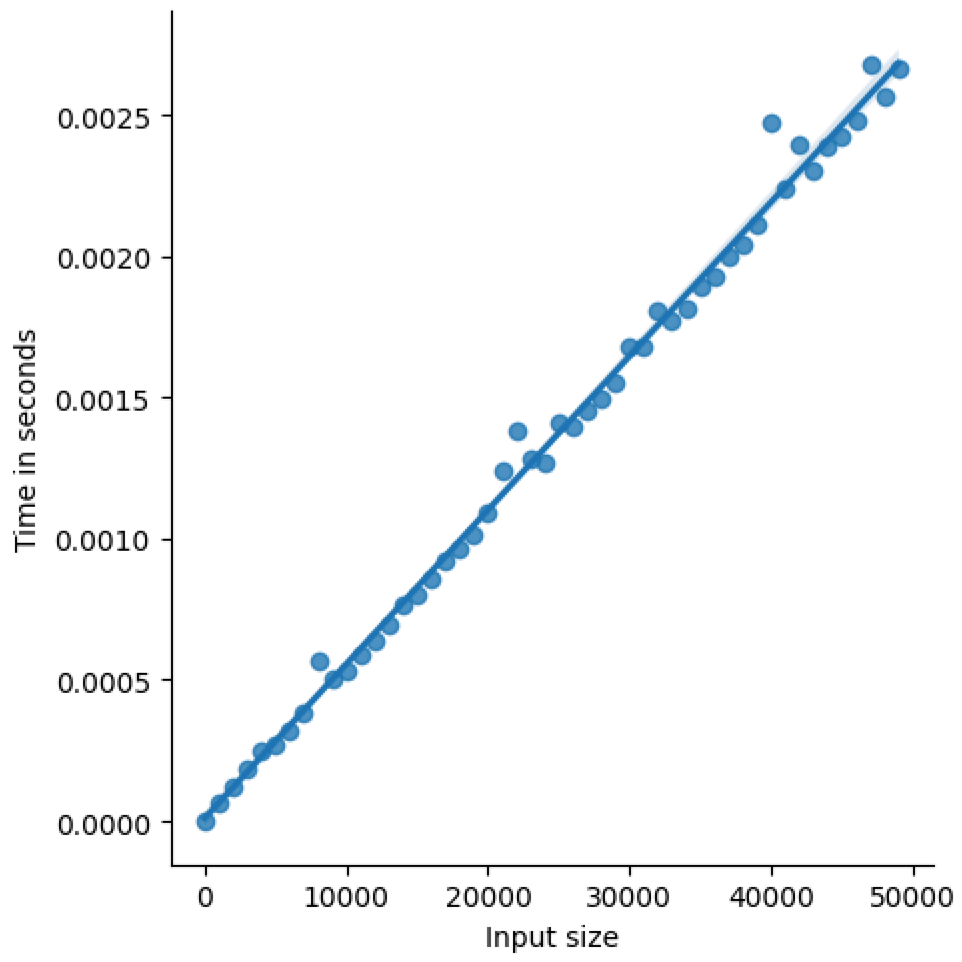
In [2]:
```python
def minimum(numbers):
    min_number = float("inf")     # Called 1 time
    for number in numbers:
        if number < min_number:   # Called n times
            min_number = number   # Called n times
    return min_number
```

- Total number of operations: $2n + 1$
- Drop lower-order terms and constant factors $\rightarrow n$
- **Time complexity:** $O(n)$

  $\rightarrow$ Runtime increases **linearly** with length of the input (n)

In [3]:

```
random_numbers = [random.randint(0, 100) for _ in range(50000)]
utils.plot_time_complexity(minimum, random_numbers, regression_order=1)
```



In [4]:
```python
def optimized_minimum(numbers):
    min_number = float("inf")          # Called 1 time
    for number in numbers:
        if number == -float("inf"):   # Called n times (worst case)
            return number
        if number < min_number:       # Called n times (worst case)
            min_number = number       # Called n times (worst case)
    return min_number
```

In [4]:

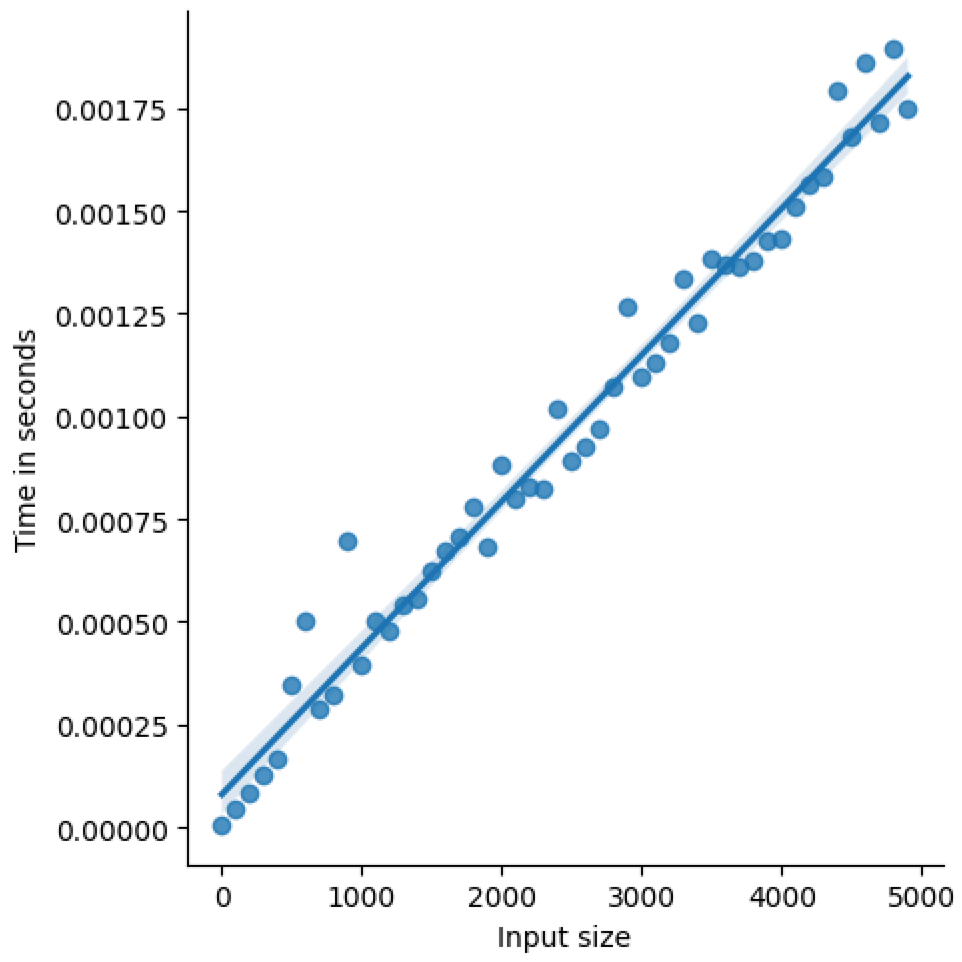
```
def optimized_minimum(numbers):
    min_number = float("inf")          # Called 1 time
    for number in numbers:
        if number == -float("inf"):  # Called n times (worst case)
            return number
        if number < min_number:        # Called n times (worst case)
            min_number = number        # Called n times (worst case)
    return min_number
```

- In the **best case** (if `numbers[0] == -inf`), we only have 2 operations
- But in the **worst case**, we have $3n + 1$ operations
- Big O notation always assumes the **worst case** scenario

→ Time complexity is still $O(n)$

In [5]:

```
random_numbers = [random.randint(0, 100) for _ in range(5000)]
utils.plot_time_complexity(optimized_minimum, random_numbers, regression_order=1)
```



In [6]:
```python
def pairwise_sums(numbers):
    """Calculate the sums of all possible pairs of numbers in a list."""
    sums = []                                          # Called 1 time
    for i in range(len(numbers)):
        for j in range(len(numbers)):
            sums.append(numbers[i] + numbers[j])  # Called n² times
    return sums
```

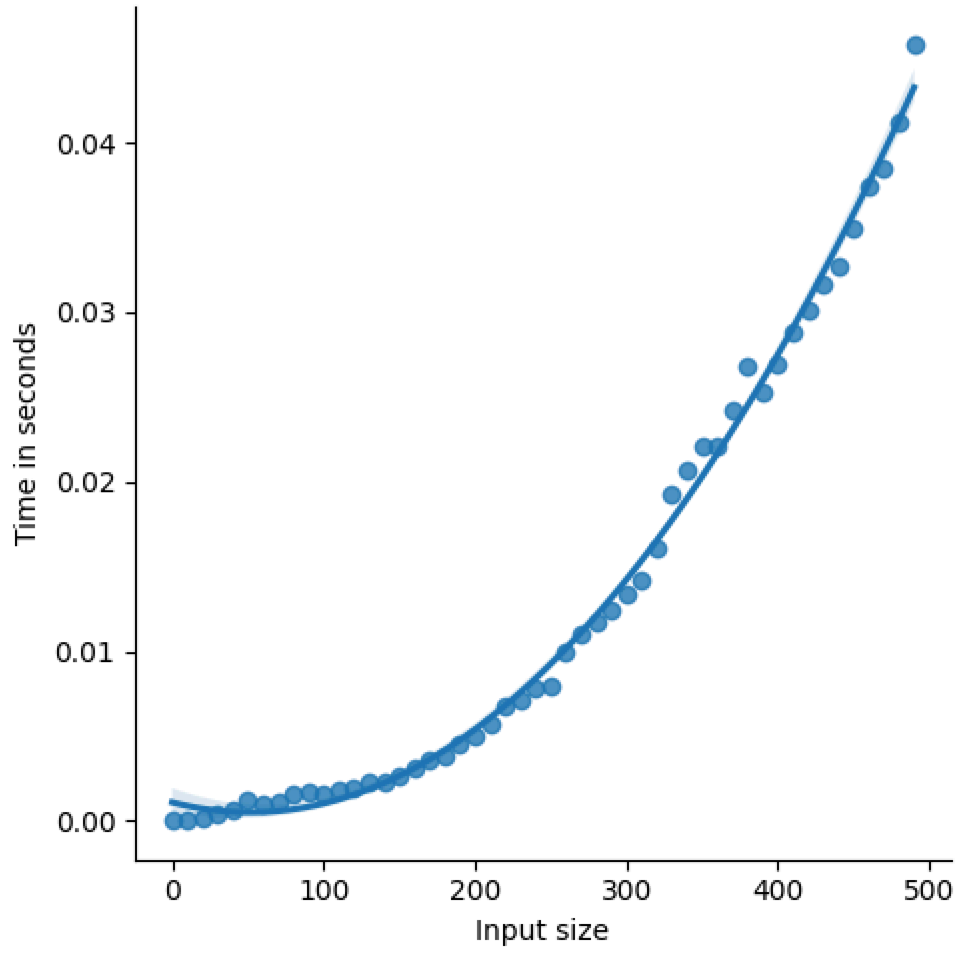
In [6]:

```python
def pairwise_sums(numbers):
    """Calculate the sums of all possible pairs of numbers in a list."""
    sums = []                                          # Called 1 time
    for i in range(len(numbers)):
        for j in range(len(numbers)):
            sums.append(numbers[i] + numbers[j])  # Called n² times
    return sums
```

- Total number of operations: $n^2 + 1$
- Drop lower-order terms and constant factors $\rightarrow$ n^2
- **Time complexity: $O(n^2)$**

  $\rightarrow$ Runtime increases **quadratically** with length of the input (n)

In [7]:
```
utils.plot_time_complexity(pairwise_sums, list(range(500)), regression_order=2)
```



In [8]:

```python
def subset_sums(numbers):
    """Calculate the sums of all possible subsets of a list."""
    sums = [0]                                     # Called 1 time
    for number in numbers:
        new_sums = []                              # Called n times
        for sum in sums:
            new_sums.append(sum + number)  # Called 2ⁿ - 1 times
        sums.extend(new_sums)                      # Called n times
    return sums
```

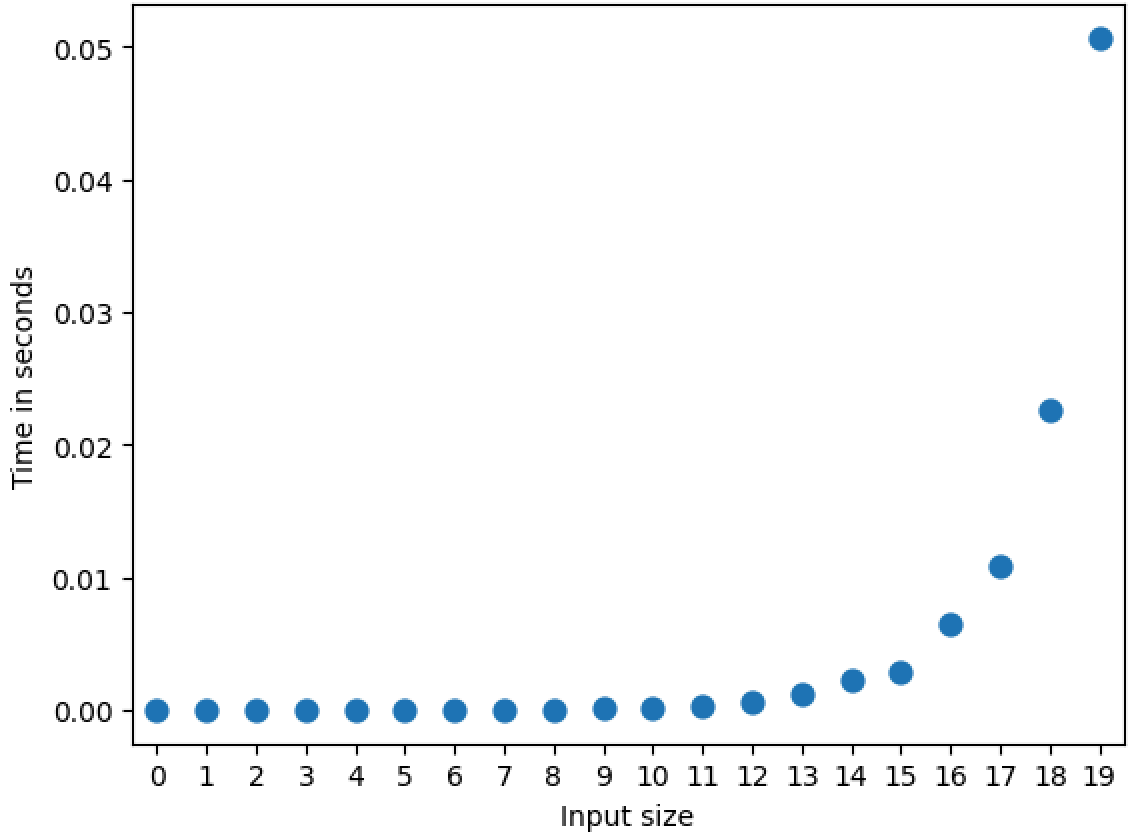
In [8]:

```python
def subset_sums(numbers):
    """Calculate the sums of all possible subsets of a list."""
    sums = [0]                                 # Called 1 time
    for number in numbers:
        new_sums = []                          # Called n times
        for sum in sums:
            new_sums.append(sum + number)  # Called 2ⁿ - 1 times
        sums.extend(new_sums)                  # Called n times
    return sums
```

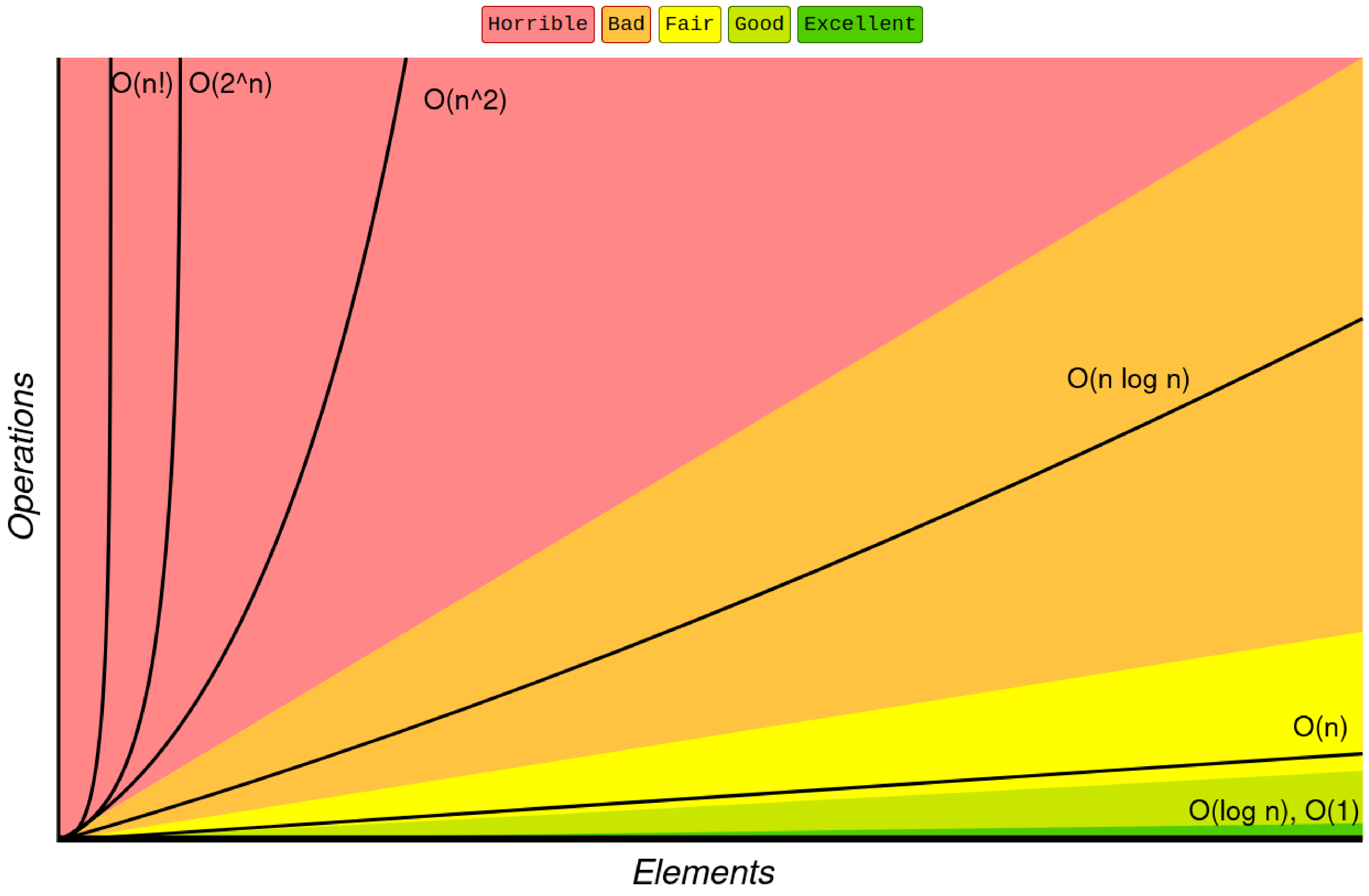
- Total number of operations: $2^n + 2n$
- Drop lower-order terms and constant factors $\rightarrow 2^n$
- **Time complexity:** $O(2^n)$

  $\rightarrow$ Runtime increases **exponentially** with length of the input (n)

In [10]:
```
utils.plot_time_complexity(subset_sums, list(range(20)))
```



Common time complexity classes



Source: bigocheatsheet.com

Remember

- We are not interested in absolute runtime (which depends on hardware)

  → Constant factors are irrelevant
- We are interested in how quickly runtime increases as inputs become very large

  → Lower-order terms become negligible

Quiz: Time complexity

pwa.klicker.uzh.ch/join/asaeub

Example: Finding duplicate strings

OpenSubtitles

- Movie subtitles in many languages
- Available in a cleaner, parallelized version as part of the OPUS corpus

  The German part can be downloaded as plain text here
- Commonly used for machine translation
- Subtitles are usually short and contain a lot of duplicates

In [11]:
```python
with open('de.txt', 'r') as f:
    lines = f.readlines()
len(lines)
```

Out[11]:
```
41612280
```

In [12]:
```python
lines[:10]
```

Out[12]:
```
['Ich geh lieber wieder an die Arbeit.\n',
 'Verspielt nicht alle Streichhölzer...\n',
 '- Hallo, Mac.\n',
 '- Hallo, Click.\n',
 'Tag, zusammen.\n',
 '- Hallo.\n',
 '- Hallo.\n',
 'Willkommen zu Hause, Mann.\n',
 'Komm, setz dich und spiel uns was vor.\n',
 '- Wir zahlen mit Versprechen.\n']
```

A naive approach

In [13]:
```python
def get_duplicates_naive(lines):
    duplicates = set()
    for i1, line1 in enumerate(lines):
        for i2, line2 in enumerate(lines):
            if line1 == line2 and i1 != i2:
                duplicates.add(line1)
    return duplicates
```

In [14]:
```python
get_duplicates_naive(lines[:500])
```

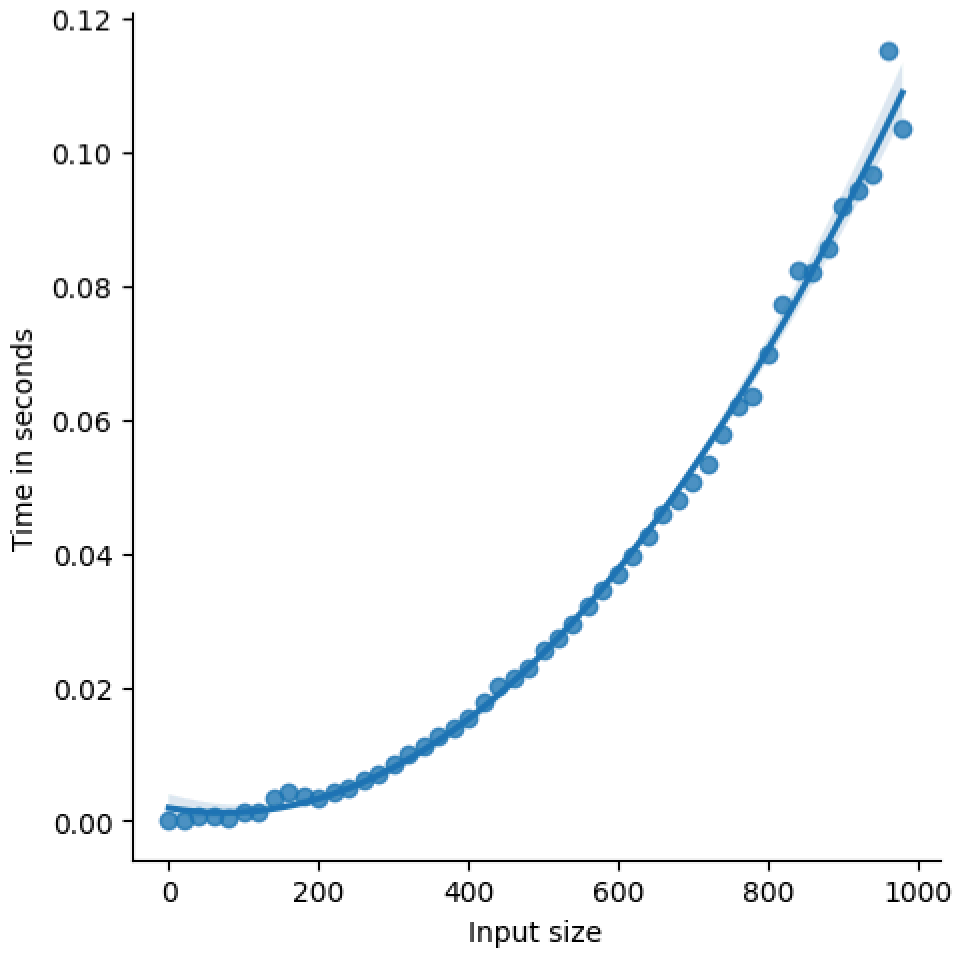
Out[14]:
```
{'- Gut.\n',
 '- Hallo.\n',
 '- Ja.\n',
 '- Morgen.\n',
 '- Nein.\n',
 '- Und wenn?\n',
 'Danke.\n',
 'Grant.\n',
 'Hier.\n',
 'Ja.\n',
 'Lass mich los!\n',
 'Nein.\n',
 'Weit reisen kannst du nur auf Gleisen\n',
 'Wieso?\n',
 'Wirklich?\n'}
```

In [16]:
```python
timeit.timeit(lambda: get_duplicates_naive(lines[:10000]), number=1)
```

Out[16]:
```
2.4275523179999254
```

In [17]:

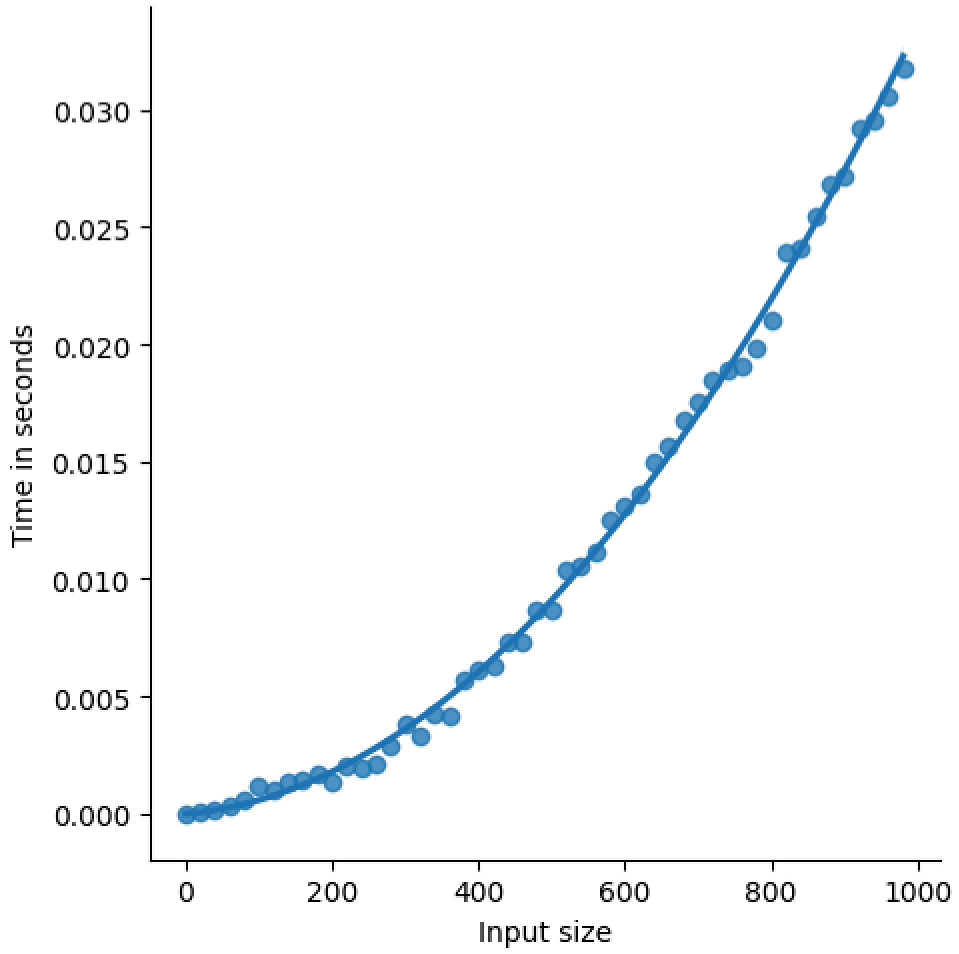
```
utils.plot_time_complexity(get_duplicates_naive, lines[:1000], regression_order=2)
```



A better approach?

In [18]:
```
def get_duplicates_maybe_better(lines):
    duplicates = set()
    for line in lines:
        count = lines.count(line)
        if count > 1:
            duplicates.add(line)
    return duplicates
```

In [19]:
```
utils.plot_time_complexity(get_duplicates_maybe_better, lines[:1000], regression_order=2)
```

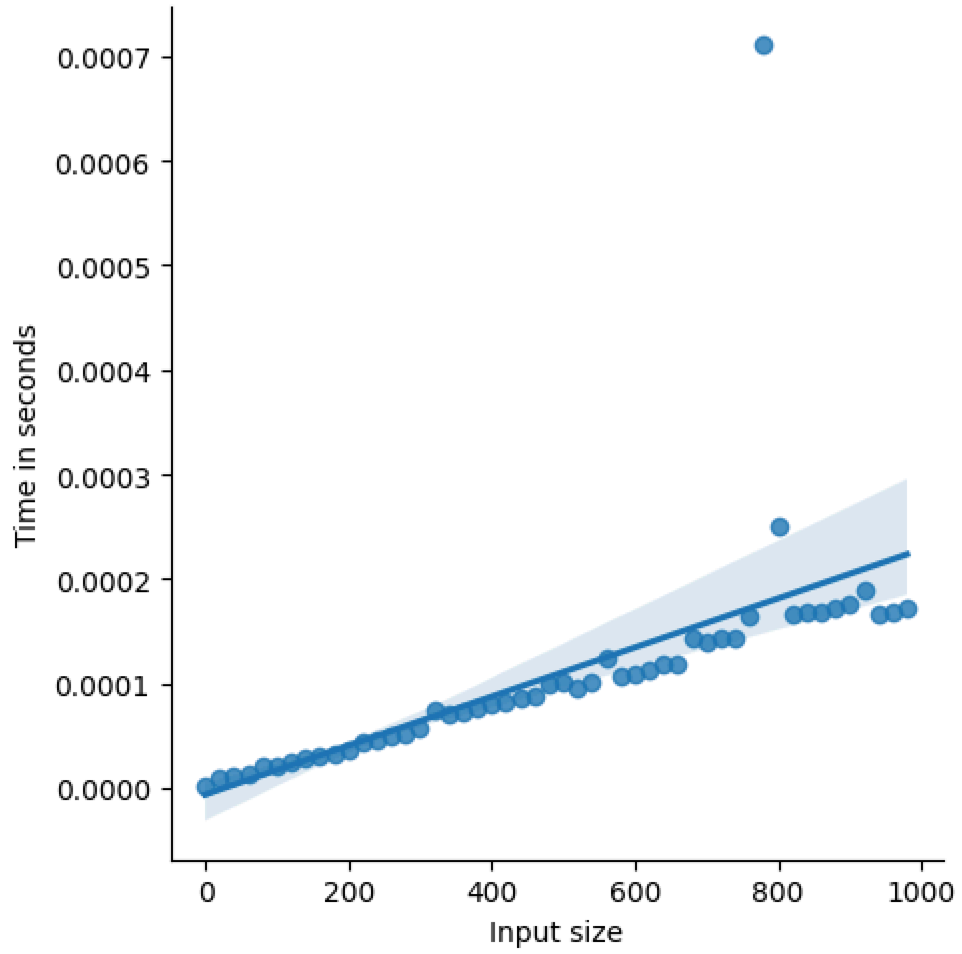


Actually a better approach

In [20]:

```
def get_duplicates_really_better(lines):
    lines_set = set()
    duplicates = set()
    for line in lines:
        if line in lines_set:
            duplicates.add(line)
        else:
            lines_set.add(line)
    return duplicates
```

In [23]:
```
utils.plot_time_complexity(get_duplicates_really_better, lines[:1000], regression_order=1)
```



Time complexity in `list`s

Due to the way `list` is implemented in Python, the following methods need to iterate over all elements (in the worst case):

- `list.count()`
- `list.index()`
- `list.__contains__()`

Their time complexity is $O(n)$ (= linear).

Overview: Time complexity in `list`s

Method	Time complexity
`append(x)`	$O(1)$
`__getitem__(i)`	$O(1)$
`__len__()`	$O(1)$
`pop()`	$O(1)$
`pop(0)`	$O(n)$
`remove(x)`	$O(n)$
`insert(i, x)`	$O(n)$
`__contains__(x)`	$O(n)$
`count(x)`	$O(n)$
`reverse()`	$O(n)$
`sort()`	$O(n \log n)$

More details: wiki.python.org/moin/TimeComplexity

Time complexity in `dict`s and `set`s

`dict` and `set` are implemented using **hash tables**. These are very efficient for looking up values:

- `set.__contains__()`
- `dict.__getitem__()`

These methods have time complexity $O(1)$ (= constant).

Overview: Time complexity in `set`s

Method	Time complexity
`add(x)`	$O(1)^*$
`pop()`	$O(1)$
`__len__()`	$O(1)$
`__contains__()`	$O(1)$

Overview: Time complexity in `dict`s

Method	Time complexity
`__setitem__(x)`	$O(1)^*$
`__getitem__(x)`, `get(x)`	$O(1)$
`pop()`	$O(1)$
`__len__()`	$O(1)$
`__contains__()`	$O(1)$

* assuming no hash collisions

Space complexity

- Big O notation can also be used for **memory usage**
- Same principle: we look at the implementation of the algorithm and figure out how much memory is used in the **worst case** (not by running the code)

Example: Finding the k longest strings

In [ ]:

```
def longest_naive(strings, k=3):
    return sorted(strings, key=len)[-k:]

longest_naive(['a', 'ab', 'abc', 'abcd', 'abcde'])
```

Example: Finding the k longest strings

In [ ]:
```
def longest_naive(strings, k=3):
    return sorted(strings, key=len)[-k:]

longest_naive(['a', 'ab', 'abc', 'abcd', 'abcde'])
```

- `sorted()` creates a new list of size n
- The return value is a list of size k
- **Space complexity:** $O(n + k)$
- Time complexity: $O(n \log n)$

In [ ]:

```python
def longest_better(strings, k=3):
    longest = []
    for string in strings:
        if len(longest) < k:
            longest.append(string)
        else:
            shortest_longest = min(longest, key=len)
            if len(string) > len(shortest_longest):
                longest.remove(shortest_longest)
                longest.append(string)
    return longest

longest_better(['a', 'ab', 'abc', 'abcd', 'abcde'])
```

In [ ]:

```python
def longest_better(strings, k=3):
    longest = []
    for string in strings:
        if len(longest) < k:
            longest.append(string)
        else:
            shortest_longest = min(longest, key=len)
            if len(string) > len(shortest_longest):
                longest.remove(shortest_longest)
                longest.append(string)
    return longest

longest_better(['a', 'ab', 'abc', 'abcd', 'abcde'])
```

- The auxiliary list `longest` has size k
- The return value has size k
- Everything else requires only constant space
- **Space complexity:** $O(k)$
- Time complexity: ?

Recursive functions

Problem: Calculate the sum of numbers in arbitrarily nested data structures like this:

In [24]:
```
data = [1, 2, [3, 4], 5, [6, [7, 8]]]
```

Recursive functions

Problem: Calculate the sum of numbers in arbitrarily nested data structures like this:

In [24]:
```
data = [1, 2, [3, 4], 5, [6, [7, 8]]]
```

This won't work:

In [25]:
```
sum(data)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 1
----> 1 sum(data)

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Recursive functions

Problem: Calculate the sum of numbers in arbitrarily nested data structures like this:

In [24]:
```
data = [1, 2, [3, 4], 5, [6, [7, 8]]]
```

This won't work:

In [25]:
```
sum(data)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 1
----> 1 sum(data)

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Solution: Recursively sum up elements of nested lists:

In [26]:

```python
def deepsum(data):
    total = 0
    for item in data:
        if isinstance(item, list):
            total += deepsum(item)  # Recursive call
        else:
            total += item           # Termination
    return total
```

In [27]:

```python
deepsum([1, 2, [3, 4], 5, [6, [7, 8]]])
```

Out[27]:

```
36
```

In [26]:
```python
def deepsum(data):
    total = 0
    for item in data:
        if isinstance(item, list):
            total += deepsum(item)  # Recursive call
        else:
            total += item           # Termination
    return total
```

In [27]:
```python
deepsum([1, 2, [3, 4], 5, [6, [7, 8]]])
```

Out[27]:
```
36
```

How many times was `deepsum` called?

In [28]:

```python
call_counter = utils.CallCounter()

@call_counter.register
def deepsum(data):
    total = 0
    for item in data:
        if isinstance(item, list):
            total += deepsum(item)
        else:
            total += item
    return total

deepsum([1, 2, [3, 4], 5, [6, [7, 8]]])
call_counter.print_most_common()
```

```
1       deepsum([1, 2, [3, 4], 5, [6, [7, 8]]])
1       deepsum([3, 4])
1       deepsum([6, [7, 8]])
1       deepsum([7, 8])
```

Recursion tree

```
deepsum([1, 2, [3, 4], 5, [6, [7, 8]]])

deepsum([3, 4])          deepsum([6, [7, 8]])

                              deepsum([7, 8])
```

What about the time complexity of `deepsum`?

The deeper the data structure, the longer the runtime:

- `deepsum([[1], [[2], [[3]]]])` takes longer than `deepsum([1, 2, 3])`

The broader the data structure, the longer the runtime:

- `deepsum([1, 2, 3, 4, 5, 6])` takes longer than `deepsum([1, 2, 3])`

→ Runtime depends on number of elements and depth: $O(n \times d)$

Levenshtein distance

How to turn `zebra` into `amoeba`?

- **Edit operations**: we can *insert*, *delete*, or *replace* letters
- Every edit operation comes with a **cost**
- The **edit distance** is the smallest possible cost to get from word A to word B
- The most common variant is the **Levenshtein distance** and defines:

Edit operation	Cost
Insertion	1
Deletion	1
Substitution	1

→ Levenshtein distance = number of edit operations

`zebra` → `amoeba`: naive approach

1. Replace `z` with `a` → costs 1
2. Replace `e` with `m` → costs 1
3. Replace `b` with `o` → costs 1
4. Replace `r` with `e` → costs 1
5. Replace `a` with `b` → costs 1
6. Insert `a` → costs 1

`zebra` → `amoeba`: naive approach

1. Replace `z` with `a` → costs 1
2. Replace `e` with `m` → costs 1
3. Replace `b` with `o` → costs 1
4. Replace `r` with `e` → costs 1
5. Replace `a` with `b` → costs 1
6. Insert `a` → costs 1

Total cost: 6 → Can we do better?

`zebra` → `amoeba`: optimal solution

1. Replace `z` with `a` → costs 1
2. Insert `m` → costs 1
3. Insert `o` → costs 1
4. Keep `e`
5. Keep `b`
6. Delete `r` → costs 1
7. Keep `a`

`zebra` → `amoeba`: optimal solution

1. Replace `z` with `a` → costs 1
2. Insert `m` → costs 1
3. Insert `o` → costs 1
4. Keep `e`
5. Keep `b`
6. Delete `r` → costs 1
7. Keep `a`

Total cost: 4 (= Levenshtein distance)

Quiz: Levenshtein distance

pwa.klicker.uzh.ch/join/asaeub

A convenient property of the Levenshtein distance problem

We can derive the Levenshtein distance of the **full strings** from the Levenshtein distance between some **substrings**.

A convenient property of the Levenshtein distance problem

We can derive the Levenshtein distance of the **full strings** from the Levenshtein distance between some **substrings**.

For example, if we already know the following:

- **levenshtein(`zebra` → `amoeb`) = 5**
- **levenshtein(`zebr` → `amoeba`) = 4**
- **levenshtein(`zebr` → `amoeb`) = 4**

Then we can easily get **levenshtein(`zebra` → `amoeba`)**.

1. Suppose we already know that **levenshtein(`zebra` → `amoeb`) = 5**
   → Turning `zebra` into `amoeba` is possible with **1 additional edit operation** (inserting `a`)
   → Total cost: **6**

1. Suppose we already know that **levenshtein(`zebra` → `amoeb`) = 5**
   → Turning `zebra` into `amoeba` is possible with **1 additional edit operation** (inserting `a`)
   → Total cost: **6**

2. Suppose we already know that **levenshtein(`zebr` → `amoeba`) = 4**
   → Turning `zebra` into `amoeba` is possible with **1 additional edit operation** (deleting `a`)
   → Total cost: **5**

1. Suppose we already know that **levenshtein(`zebra` → `amoeb`) = 5**
   → Turning `zebra` into `amoeba` is possible with **1 additional edit operation** (inserting `a`)
   → Total cost: **6**

2. Suppose we already know that **levenshtein(`zebr` → `amoeba`) = 4**
   → Turning `zebra` into `amoeba` is possible with **1 additional edit operation** (deleting `a`)
   → Total cost: **5**

3. Suppose we already know that **levenshtein(`zebr` → `amoeb`) = 4**
   → Turning `zebra` into `amoeba` is possible **without additional edit operations** (keeping `a`)
   → Total cost: **4**

1. Suppose we already know that **levenshtein(`zebra` → `amoeb`) = 5**
   → Turning `zebra` into `amoeba` is possible with **1 additional edit operation** (inserting `a`)
   → Total cost: **6**

2. Suppose we already know that **levenshtein(`zebr` → `amoeba`) = 4**
   → Turning `zebra` into `amoeba` is possible with **1 additional edit operation** (deleting `a`)
   → Total cost: **5**

3. Suppose we already know that **levenshtein(`zebr` → `amoeb`) = 4**
   → Turning `zebra` into `amoeba` is possible **without additional edit operations** (keeping `a`)
   → Total cost: **4**

Solution 3 is the cheapest, and there are no other solutions.

Therefore, **levenshtein(`zebra` → `amoeba`) = 4**

Recursive definition of Levenshtein distance

$$
\operatorname{levenshtein}(a, b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{levenshtein}\big(a[:-1], b[:-1]\big) & \text{if } a[-1] = b[-1], \\ 1 + \min \begin{cases} \operatorname{levenshtein}\big(a, b[:-1]\big) \\ \operatorname{levenshtein}\big(a[:-1], b\big) \\ \operatorname{levenshtein}\big(a[:-1], b[:-1]\big) \end{cases} & \text{otherwise} \end{cases}
$$

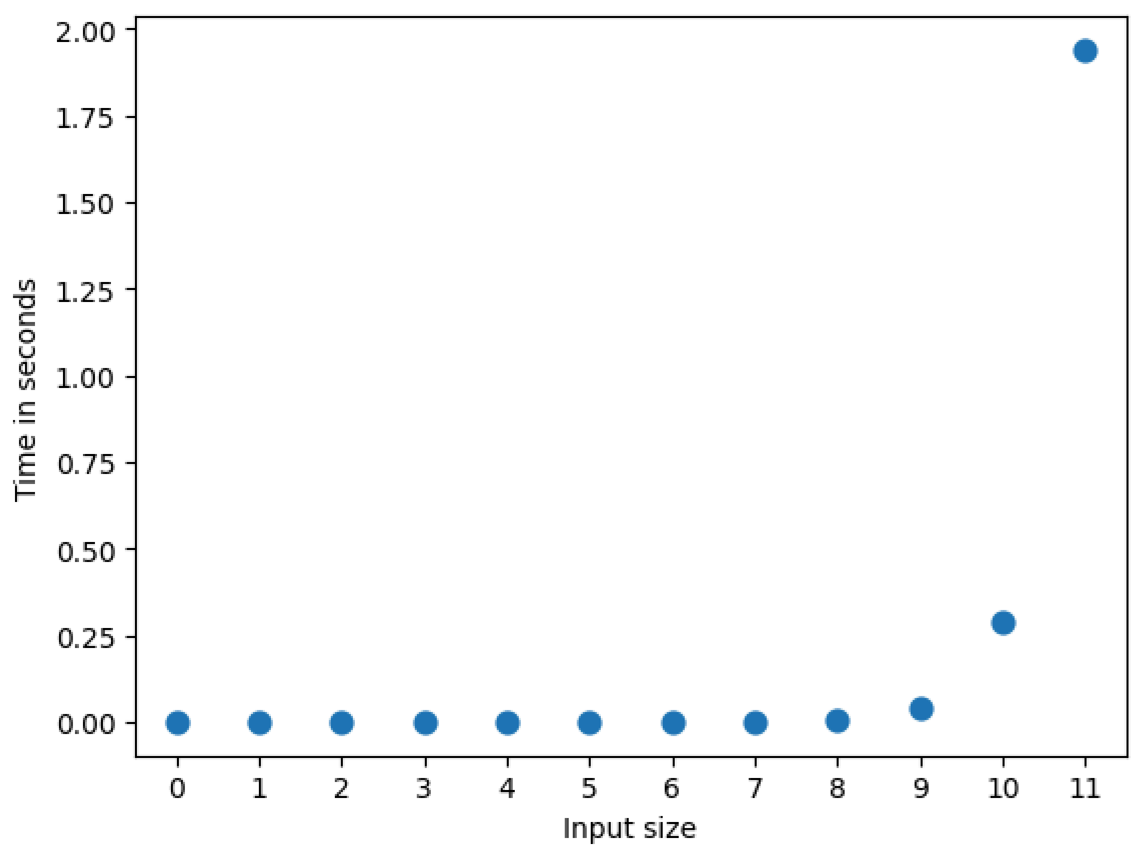
In [29]:
```python
def levenshtein(a: str, b: str) -> int:
    if a == "":
        return len(b)                               # Termination
    if b == "":
        return len(a)                               # Termination
    if a[-1] == b[-1]:
        return levenshtein(a[:-1], b[:-1])  # Recursive call
    return 1 + min(
        levenshtein(a, b[:-1]),                     # Recursive call
        levenshtein(a[:-1], b),                     # Recursive call
        levenshtein(a[:-1], b[:-1]),                # Recursive call
    )

levenshtein("zebra", "amoeba")
```

Out[29]: 4

What is the time complexity of the recursive Levenshtein distance algorithm?

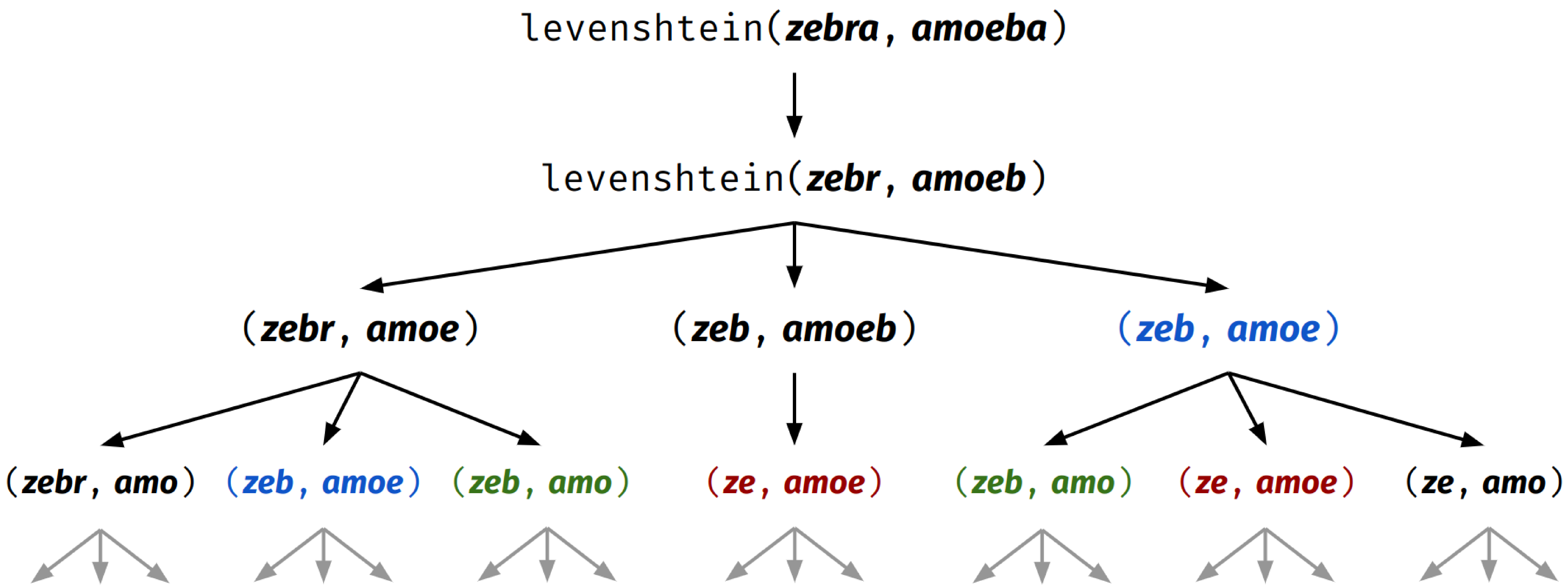
In [30]:
```python
random_string = "".join(random.choices(string.ascii_letters, k=12))
utils.plot_time_complexity(lambda x: levenshtein(x, "".join(reversed(x))), random_string, number=1)
```



Why is it so bad?

In [31]:
```
call_counter = utils.CallCounter()
levenshtein = call_counter.register(levenshtein)
levenshtein("zebra", "amoeba")
call_counter.print_most_common()
```

```
108     levenshtein('', 'a')
107     levenshtein('z', '')
77      levenshtein('z', 'a')
77      levenshtein('', '')
40      levenshtein('', 'am')
38      levenshtein('ze', '')
31      levenshtein('z', 'am')
30      levenshtein('ze', 'a')
16      levenshtein('ze', 'am')
9       levenshtein('zeb', '')
9       levenshtein('z', 'amo')
9       levenshtein('', 'amo')
8       levenshtein('zeb', 'a')
6       levenshtein('zeb', 'am')
6       levenshtein('ze', 'amo')
4       levenshtein('zeb', 'amo')
3       levenshtein('ze', 'amoe')
2       levenshtein('zeb', 'amoe')
1       levenshtein('zebra', 'amoeba')
1       levenshtein('zebr', 'amoeb')
1       levenshtein('zebr', 'amoe')
1       levenshtein('zebr', 'amo')
1       levenshtein('zebr', 'am')
```


levenshtein(zebra, amoeba)
levenshtein(zebr, amoeb)
(zebr, amoe)
(zeb, amoeb)
(zeb, amoe)
(zebr, amo)
(zeb, amoe)
(zeb, amo)
(ze, amoe)
(zeb, amo)
(ze, amoe)
(ze, amo)

More efficient implementation of Levenshtein distance

(See `levenshtein.pdf` or video on OLAT)

Good online demo: https://phiresky.github.io/levenshtein-demo/

In [32]:
```python
def levenshtein_recursive(a: str, b: str) -> int:
    """Return the Levenshtein distance between two strings using recursion."""
    if a == "":
        return len(b)
    if b == "":
        return len(a)
    if a[-1] == b[-1]:
        return levenshtein_recursive(a[:-1], b[:-1])
    return 1 + min(
        levenshtein_recursive(a, b[:-1]),
        levenshtein_recursive(a[:-1], b),
        levenshtein_recursive(a[:-1], b[:-1]),
    )

levenshtein_recursive("zebra", "amoeba")
```

Out[32]:

```
4
```

In [33]:

```python
def levenshtein_dynamic(a: str, b: str) -> int:
    """Return the Levenshtein distance between two strings using dynamic programming."""
    # Initialize table
    table = [[0] * (len(b) + 1) for _ in range(len(a) + 1)]
    for i in range(len(a) + 1):
        table[i][0] = i
    for j in range(len(b) + 1):
        table[0][j] = j
    # Fill table
    for i in range(1, len(a) + 1):
        for j in range(1, len(b) + 1):
            if a[i - 1] == b[j - 1]:
                table[i][j] = table[i - 1][j - 1]  # Keep
            else:
                table[i][j] = 1 + min(
                    table[i][j - 1],      # Insert
                    table[i - 1][j],      # Delete
                    table[i - 1][j - 1],  # Replace
                )
    # Solution in the bottom right corner
    return table[-1][-1]

levenshtein_dynamic("zebra", "amoeba")
```

Out[33]:

```
4
```

Dynamic programming

- This tabular approach of finding the edit distance is an example of **dynamic programming**
- **Some recursive problems** can be solved more efficiently using dynamic programming

Dynamic programming

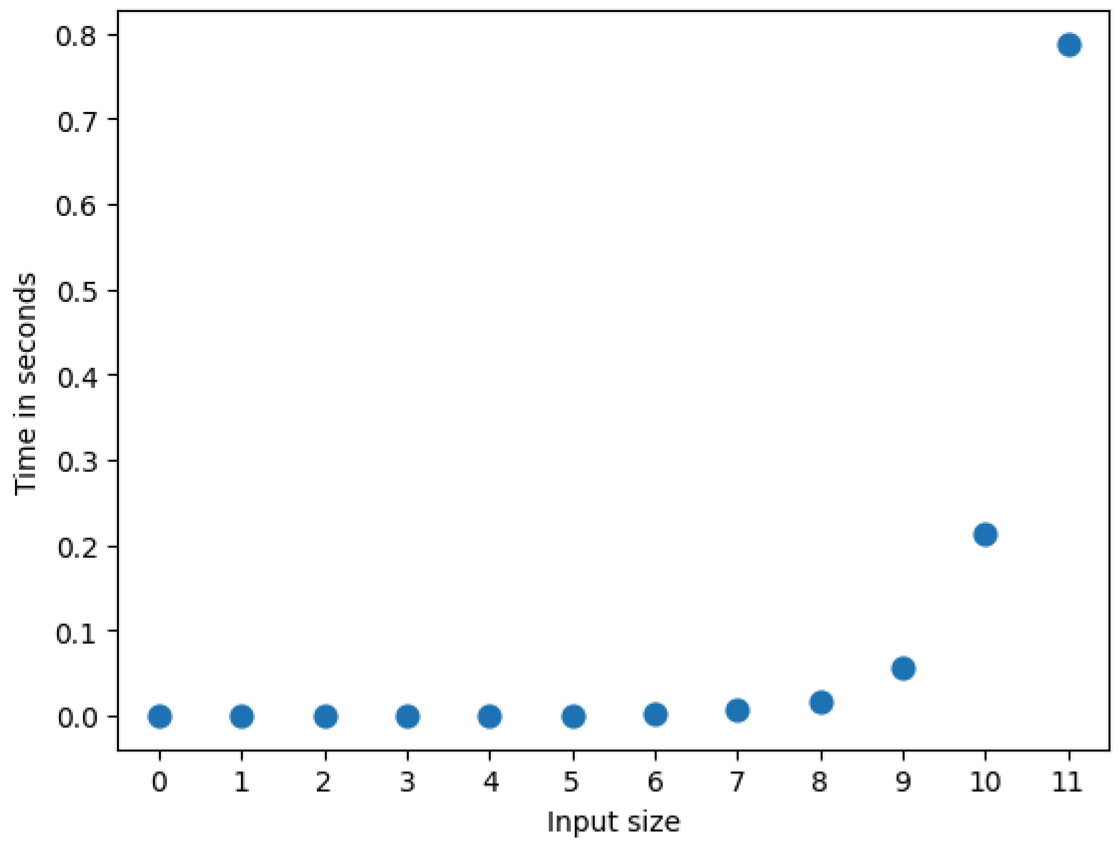
- This tabular approach of finding the edit distance is an example of **dynamic programming**
- **Some recursive problems** can be solved more efficiently using dynamic programming

Requirements for applying dynamic programming:

- The problem can be divided into **subproblems**
  Example: Edit distance between strings > edit distance between substrings
- The optimal solution for the problem can be **derived from optimal solutions for the subproblems**
  Example: If we know the edit distance between all substrings, we know the edit distance between the full strings

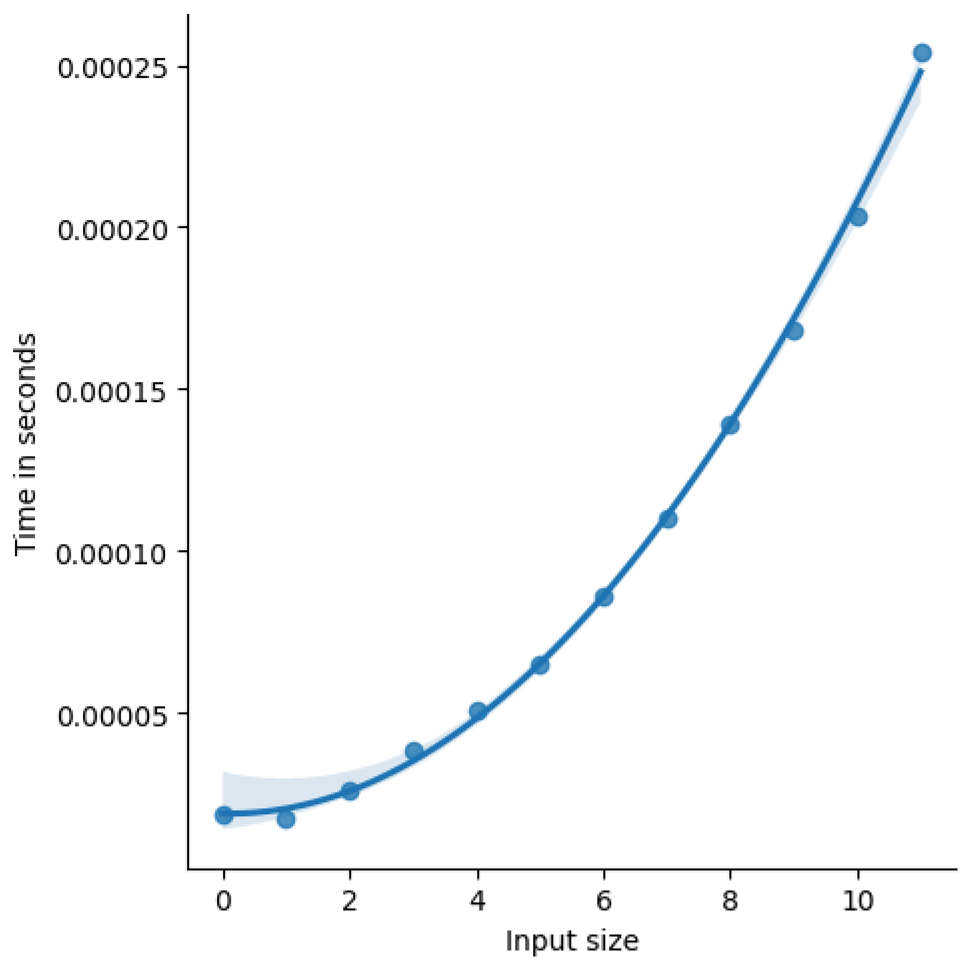
Complexity without dynamic programming

In [34]:
```
random_string = "".join(random.choices(string.ascii_letters, k=12))
utils.plot_time_complexity(lambda x: levenshtein_recursive(x, "".join(reversed(x))), random_string,
```



Time complexity with dynamic programming

In [35]:
```
random_string = "".join(random.choices(string.ascii_letters, k=12))
utils.plot_time_complexity(lambda x: levenshtein_dynamic(x, "".join(reversed(x))), random_string, r
```



More examples of dynamic programming

- Text-to-speech: Viterbi algorithm for finding the best speech samples in context
- Syntax parsing: CYK algorithm for context-free grammar parsing
- Graphs (e.g., WordNet): Dijkstra's algorithm for finding the shortest path between two nodes
- Sequence alignment (similar to edit distance!): matching DNA or protein sequences

Take-home messages

- **Computational complexity** measures how efficient an algorithm is as the size of its input increases
  - **Time complexity** and **space complexity**
  - $O(1) < O(n) < O(n \log n) < O(n^2) < O(2^n)$
  - Complexity is a theoretical concept -- it doesn't tell us anything about how many seconds or bytes the algorithm will take to run!
- Checking if a specific value exists in a `list` is slow! Use `set` or `dict` instead
- **Recursive functions** are functions that call themselves
- **Dynamic programming** is a technique to reduce time complexity by dividing the problem into subproblems and storing the results of those subproblems
- **Levenshtein distance** is the lowest number of edit operations (insertions, deletions, substitutions) required to turn one string into another

Enjoy your spring break! :)