When an element is added to the end of a list, the time needed is $O(1)$ or $O(n)$ depending on whether the memory area reserved for the list has room for the new element or not. If there is no room, a new, bigger memory area is reserved and all the elements are copied there from the old memory area.

Although the *worst case* time complexity is $O(n)$, it can be shown that the *average case* time complexity is $O(1)$ with an appropriate method of memory reservation. One such method is to *double* the reserved memory area whenever more memory is needed.

Consider a situation, where the list contains n elements and it was just moved to a new memory area. That last relocation moved n elements, the one before that moved $n/2$ elements, the one before that moved $n/4$ elements, etc.. Thus the total number of element moves is:

$$n + n/2 + n/4 + n/8 + \cdots < 2n = O(n).$$

Because n elements have been added to the list and the total number of moves is $O(n)$, the number of moves per element addition is $O(1)$ *on average*.

More generally, if the size of the memory area is multiplied by c with each expansion, the number of moves is

$$n + n/c + n/c^2 + n/c^3 + \cdots = O(n),$$

where c is any constant bigger than 1. The constant c controls a tradeoff between efficiency and memory use. A bigger c means that there is less copying and more memory reserved but unused.

Python implementation

We can study the memory use of a Python list with the following code:

```python
import sys

n = 100

numbers = []
old_size = 0

for i in range(n):
    new_size = sys.getsizeof(numbers)

    if new_size != old_size:
        print(len(numbers), new_size)
        old_size = new_size

    numbers.append(1)
```

The code uses the function `sys.getsizeof` that returns the memory used by the given object in bytes. The code creates an empty list and then adds elements to the list one at a time. Whenever the memory usage changes, the code prints out the length and memory use of the list.

In the test computer (CPython 3.10.6), the code prints:

```
0 56
1 88
5 120
9 184
17 248
25 312
33 376
41 472
53 568
65 664
77 792
93 920
```

This shows that an empty list requires 56 bytes of memory, and that each additional element needs 8 bytes. The memory usage grows when the number of elements grows to 1, 5, 9, 17, 25, etc.. For example, when the element count reaches 17, the new memory usage is 248 bytes and there is room for (248 - 56) / 8 = 24 elements. Thus the next expansion happens when the element count reaches 25.

Studying the list implementation in CPython shows that the number of elements that fit in the new memory area is $n + \lfloor n/8 \rfloor + 6$ rounded down to the nearest multiple of 4, where n is the element count that triggered the expansion. For example when $n = 17$, the formula evaluates to $17 + \lfloor 17/8 \rfloor + 6 = 25$ and the nearest multiple of 4 is 24. This means that the memory size changes approximately by the factor $9/8$ in each expansion.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI