3.4 Analysing Program Running Time

List - based implementation of the Stack ADT:

Using the front of the list as the top of the stack

```
class Stack2:
    """Alternate stack implementation.

    This implementation uses the *front* of the Python list to represent
    the top of the stack.
    """
    # === Private Attributes ===
    # _items:
    #     The items stored in the stack. The front of the list represents
    #     the top of the stack.
    _items: List

    def __init__(self) -> None:
        """Initialize a new empty stack."""
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this stack contains no items.

        >>> s = Stack()
        >>> s.is_empty()
        True
        >>> s.push('hello')
        >>> s.is_empty()
        False
        """
        return self._items == []

    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack."""
```

```
        self._items.insert(0, item)

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Raise an EmptyStackError if this stack is empty.

        >>> s = Stack()
        >>> s.push('hello')
        >>> s.push('goodbye')
        >>> s.pop()
        'goodbye'
        """
        if self.is_empty():
            raise EmptyStackError
        else:
            return self._items.pop(0)
```

TIMEIT:

- Rough estimate of how long it takes code to run

```python
def push_and_pop(s: Stack) -> None:
    """Push and pop a single item onto <stack>.

    This is simply a helper for the main timing experiment.
    """
    s.push(1)
    s.pop()


if __name__ == '__main__':
    # Import the main timing function.
    from timeit import timeit

    # The stack sizes we want to try.
    STACK_SIZES = [1000, 10000, 100000, 1000000, 10000000]
    for stack_size in STACK_SIZES:
        # Uncomment the stack implementation that we want to time.
        stack = Stack()
        # stack = Stack2()

        # Bypass the Stack interface to create a stack of size <stack_size>.
        # This speeds up the experiment, but we know this violates encapsulation!
        stack._items = list(range(stack_size))

        # Call push_and_pop(stack) 1000 times, and store the time taken in <time>.
        # The globals=globals() is used for a technical reason that you can ignore.
        time = timeit('push_and_pop(stack)', number=1000, globals=globals())

        # Finally, report the result. The :>8 is used to right-align the stack size
        # when it's printed, leading to a more visually-pleasing report.
        print(f'Stack size {stack_size:>8}, time {time}')
```

MEMORY ALLOCATION FOR LISTS IN PYTHON:

- They are an object that contains an ordered sequence of references to other objects
- List must be continuous
- Insertion and deletion is less efficient as all items have to be moved up/down by one block of memory.
- Tradeoff : give up fast insertion and deletion for fast lookup by index!
- Python allocates more memory to the end of a list than it needs which is why its faster to add/remove items AT THE END OF A LIST!

ANALYSING ALGORITHMIC RUNNING TIME:

- **Correctness:**
  - Does code work even with corner cases & does it handle errors?
- **Design:**
  - Is the code easy to understand and easy to implement?

BIG OH NOTATION:

- Ignoring constants, focusing on behaviour as problem size grows (*asymptotic runtime*)

Big-Oh	Growth term
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(2^n)$	exponential (with base 2)

Big-Oh	Relationship
$O(\log n)$	$N_1 \approx N_0 + c$
$O(n)$	$N_1 \approx 2N_0$
$O(n^2)$	$N_1 \approx 4N_0$
$O(2^n)$	$N_1 \approx (N_0)^2$