Algorithm Analysis

What is an algorithm

An algorithm is a set of instructions that are used to solve a problem.

Example

Find the maximum value in a list of numbers.

1. Set the maximum value to the first value in the list.
2. For each value in the list, if the value is greater than the maximum value, then set the maximum value to that value.
3. Return the maximum value after looking at all values in the list.

How can we compare algorithms?

- **Time complexity** - How long does it take to run the algorithm?
- **Space complexity** - How much memory does the algorithm use?
- **Correctness** - Does the algorithm solve the problem, or does it approximate the solution?

Big O Notation

One way to compare algorithms is by understanding its behavior as the size of the problem increases. Big O notation is used to describe the time complexity of an algorithm.

We say an algorithm has a time complexity $O(f(n))$ if the number of operations is bounded by $Cf(n)$ for some constant C and for all n greater than some constant n_0.

$$O(f(n)) = \{g(n) : \exists C > 0, \exists n_0 > 0, \forall n > n_0, 0 \leq g(n) \leq Cf(n)\}$$

They normally considered the amount of stpes that the algorithm has to perform in the worst case scenario. E.g. sorting a list that is in reverse order.

Examples of Big O Notation

- $O(1)$ - Constant time, and algorithm that always takes the same amount of time to run. E.g. accessing an element in an array.

```
a = range(1000000)
%timeit a[0] # O(1)
%timeit a[500000] # O(1)
```

Differences are due to CPU caching, practically they are the same.

```
66.8 ns ± 0.124 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
86.2 ns ± 0.192 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

Examples of Big O Notation (2)

- $O(n)$ - Linear time, an algorithm that takes n steps to run. E.g. find the maximum value in a list of numbers.

```
import random
random.seed(0)
a = [random.random() for _ in range(1000)]
%timeit max(a) # O(n)
a = [random.random() for _ in range(1000000)]
%timeit max(a) # O(n)
```

Second examples takes 1000 times longer.

$1ms = 1000\mu s$

```
12.1 µs ± 4.11 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
12 ms ± 10.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Examples of Big O Notation (3)

- $O(n^2)$ - Quadratic time, an algorithm that takes n^2 steps to run. Sort a list of numbers using bubble sort.

```
import random
random.seed(0)
a = [random.random() for _ in range(1000)]
%timeit bubble_sort(a) # O(n^2)
a = [random.random() for _ in range(10000)]
%timeit bubble_sort(a) # O(n^2)
```

Increasing the size ten times increases the time by 100 times.

```
37 ms ± 107 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
4.09 s ± 24.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

$$\frac{4.09s}{37} = 110.54$$

Appendix: Bubble Sort

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Polynomial time

When an algorithm has a time complexity of $O(n^k)$ for some constant k, we say it has polynomial time. Polynomial time algorithms are considered efficient.

Example NumPy's matrix inversion is approximately $O(n^3)$. This means that increasing the size of the matrix by 10 times increases the time by 1000 times.

DO NOT RUN IN A SLOW COMPUTER

```
import numpy as np
import random
random.seed(0)
a = np.random.rand(1000, 1000)
%timeit np.linalg.inv(a) # O(n^3)
a = np.random.rand(100000, 100000)
%timeit np.linalg.inv(a) # O(n^3)
```

Logarithmic time

When an algorithm has a time complexity of $O(\log n)$, we say it has logarithmic time. Logarithmic time algorithms are considered efficient.

Example Binary search is a search algorithm that finds the position of a target value within a sorted array. Increasing the size by 1000 barely changes the time.

```
a = range(1000000)
%timeit binary_search(a, 500000) # O(log n)
a = range(1000000000)
%timeit binary_search(a, 500000) # O(log n)
```

```
5.18 µs ± 9.36 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
7.79 µs ± 72 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Appendix: Binary Search

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] < target:
            low = mid + 1
        elif arr[mid] > target:
            high = mid - 1
        else:
            return mid
    return -1
```

Exponential time

When an algorithm has a time complexity of $O(2^n)$, we say it has exponential time. Exponential time algorithms are considered inefficient.

Example The Power Set problem involves finding all possible subsets of a given set, including the empty set and the set itself. For a set with n elements, the number of subsets is 2^n, which grows exponentially with the size of the set. An extra element almost doubles the time.

```
s = range(5)
%timeit generate_power_set(s) # O(2^n)
s = range(6)
%timeit generate_power_set(s) # O(2^n)
```

```
6.05 µs ± 18.8 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
10.7 µs ± 20.2 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Appendix: Compute the Power Set

```
def generate_power_set(s):
    if len(s) == 0:
        return [[]]  # Base case: empty set has one subset, which is the empty set

    subsets = []
    first_element = s[0]
    remaining_elements = s[1:]

    # Recursive call to generate subsets without the first element
    subsets_without_first = generate_power_set(remaining_elements)

    # Combine subsets without the first element with subsets including the first element
    for subset in subsets_without_first:
        subsets.append(subset)  # Add subset without the first element
        subsets.append([first_element] + subset)  # Add subset including the first element

    return subsets
```

Factorial time

When an algorithm has a time complexity of $O(n!)$, we say it has factorial time. Factorial time algorithms are considered inefficient.

One example of an algorithm with a time complexity of $O(n!)$ is the brute-force solution for the permutation problem. The permutation problem involves finding all possible permutations of a given set of elements.

```
s = list(range(5))
%timeit generate_permutations(s) # O(n!)
s = list(range(6))
%timeit generate_permutations(s) # O(n!)
```

```
137 µs ± 291 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
822 µs ± 587 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

$822/137 = 6$

Appendix: Compute the Permutations

```python
def generate_permutations(elements):
    permutations = []
    generate_permutations_recursive(elements, [], permutations)
    return permutations

def generate_permutations_recursive(elements, current_permutation, permutations):
    if len(elements) == 0:
        permutations.append(current_permutation)
    else:
        for i in range(len(elements)):
            remaining_elements = elements[:i] + elements[i+1:]
            new_permutation = current_permutation + [elements[i]]
            generate_permutations_recursive(remaining_elements, new_permutation, permutations)
```