Computational Complexity

METIS®

Learning Objectives

- Understand what computational complexity is, as well as its various types

- Understand how to find the complexity of an algorithm

- Understand how to simplify the complexity of an algorithm using big O notation

Chapter 1

Intro to Complexity

METIS®

Analyzing Algorithms

Data scientists use algorithms in their work

- Which algorithm is the fastest?
- Which algorithm requires the least memory to run?
- Which algorithm is the most efficient?

Algorithms can be benchmarked using computational complexity

Types of Complexity

Time complexity

- How long does an algorithm take to run?
- Measured in the number of computational steps

Space complexity

- How much memory is needed by the algorithm?

Complexity varies depending on the dataset

Best-case complexity

- What is the lower bound on the resources required?

Average-case complexity

- What is the performance across all datasets?

Worst-case complexity

- What is the upper bound on the resources required?

Chapter 2

Calculating Complexity

METIS®

Pair Problem 1

Given a list of integers between 1 and N, write a function to generate all the different pairings of these numbers. Pairings with different orders are considered to be distinct.

For example, the permutations (1, 2) and (2, 1) are two separate pairings.

Use a nested loop

```
# numbers = [1, 2, 3]

def make_pairs (numbers):
    n = len(numbers)           # 1 step
    pairs = []                             # 1
step
    for i in range(n):              # n times
        for j in range(n):    # n times
            if i != j:
                pairs.append((l[i],l[j]))
    return pairs                           # 1
step

# pairs = [(1,2), (1,3), (2,1), (2,3),
(3,1), (3,2)]

# Complexity is n^2 + 3
```

Pair Problem 2

Given a list of integers between 1 and N, write a function to generate all the different pairings of numbers that are exactly two apart. Pairings with different orders are considered to be distinct.

For example, if N = 3, the only valid permutations are (1, 3) and (3, 1).

Modify the solution ...

```
# numbers = [1, 2, 3]

def make_pairs_a (numbers):
    n = len(numbers)           # 1 step
    pairs = []                             # 1
step
    for i in range(n):              # n times
        for j in range(n):    # n times
            if abs(i-j) == 2:
                pairs.append((l[i],l[j]))
    return pairs                           # 1
step

# pairs = [(1,3), (3,1)]

# Complexity is n^2 + 3
```

Or rewrite the code

```
# numbers = [1, 2, 3]

def make_pairs_b (numbers):
    n = len(numbers)           # 1 step
    pairs = []                          # 1
step
    for i in range(n-2):       # n-2 times
        pairs.append((l[i],l[i+2]))
    for i in range(2,n):       # n-2 times
        pairs.append((l[i],l[i-2]))
    return pairs                        # 1
step

# pairs = [(1,3), (3,1)]

# Complexity is 2n - 1
```

Solution B is more efficient

N	Solution A	Solution B
2	7	3
3	12	5
5	28	9
10	103	19
20	403	39
100	10003	199

Chapter 3

Big O Notation

METIS®

Comparing Complexity

Data scientists work with big data

The most important comparison is the overall growth behavior

$$n^2 > 2n$$

Big O Notation

1. Find expression for complexity

2. Only keep the highest order (leftmost) term

3. Discard constants and coefficients

Finding Big O

$$f_A(n) = n^2 + 3$$
$$\rightarrow n^2$$
$$f_A(n) = O(n^2)$$

$$f_B(n) = 2n - 1$$
$$\rightarrow 2n$$
$$\rightarrow n$$
$$f_B(n) = O(n)$$

$$f_C(n) = n^3 + 5n^2 + 2n - 1$$
$$\rightarrow n^3$$
$$f_C(n) = O(n^3)$$

Big O Comparison

Growth	Big O
Factorial	$O(n!)$
Exponential	$O(x^n)$
Polynomial	$O(n^x)$
Linear	$O(n)$
Logarithmic	$O(\log n)$
Constant	$O(1)$