Computational Complexity

METIS®

Learning Objectives

- Describe computational complexity and its various types
- Find the complexity of an algorithm
- Simplify algorithm complexity with big O notation

Intro to Complexity

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 by 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?

Calculating Complexity

METIS®

Example: Pair Problem

Given: List of integers 1 to N

Write a function to **generate all the different pairings** of these numbers.

Numbers do not pair with themselves, but pairing order matters. The permutations

(1, 2) and (2, 1)

are two distinct pairings.

Use a Nested Loop

```python
def make_pairs(l):
    n = len(l)
    pairs = []
    for i in range(n):
        for j in range(n):
            if i != j:
                pairs.append((l[i],l[j]))
    return pairs

make_pairs([1, 2, 3])
```

```
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1),
(3, 2)]
```

Use a Nested Loop

What is the time complexity?

```
def make_pairs(l):
    n = len(l)                          ← 1 Step
    pairs = []                          ← 1 Step
    for i in range(n):
        for j in range(n):
            if i != j:
                pairs.append((l[i],l[j]))
    return pairs
```

Use a Nested Loop

What is the time complexity?

```
def make_pairs(l):
    n = len(l)
    pairs = []
    for i in range(n):
        for j in range(n):
            if i != j:
                pairs.append((l[i],l[j]))
    return pairs
```

← **n Times**

← **n Times**

Use a Nested Loop

What is the time complexity?

```
def make_pairs(l):
    n = len(l)
    pairs = []
    for i in range(n):
        for j in range(n):
            if i != j:
                pairs.append((l[i],l[j]))
    return pairs
```

← 1 Step

Use a Nested Loop

What is the time complexity?

```
def make_pairs(l):                     ← 1 Step
    n = len(l)                         ← 1 Step
    pairs = []
    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
```

Time complexity is $n^2 + 3$

Example: Modified Pair Problem

Given: List of integers 1 to N

Write a function to generate all the different pairings **that are exactly 2 apart**.

Pairing order still matters. If N=3, the only valid pairings are

(1, 3) and (3, 1).

Option A: Modify the Solution

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

make_pairs_a([1, 2, 3])
```

```
[(1, 3), (3, 1)]
```

Option B: Rewrite the Code

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

make_pairs_b([1, 2, 3])
```

```
[(1, 3), (3, 1)]
```

Option B: Rewrite the Code

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

← 1 Step

← 1 Step

← n-2 Steps

← n-2 Steps

← 1 Step

Time complexity is

$$2(n-2) + 3 = 2n - 1$$

Solution B is More Efficient

	Computation Steps	
N	Solution A	Solution B
2	7	3
3	12	5
5	28	9
10	103	19
20	403	39
100	10,003	199

Solution B is More Efficient

	Computation Steps	
N	Solution A	Solution B
2	7	3
3	12	5
5	28	9
10	103	19
20	403	39
100	10,003	199

Solution B is More Efficient

	Computation Steps	
N	Solution A	Solution B
2	7	3
3	12	5
5	28	9
10	103	19
20	403	39
100	10,003	199

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

Solution A

$$n^2 + 3 \longrightarrow O(n^2)$$

Solution B

$$2n - 1 \longrightarrow O(n)$$

More Complex Calculation

$$n^3 + 5n^2 + 2n - 1 \longrightarrow O(n^3)$$

Big O Comparison

Growth	Big O	Resource Usage
Factorial	O(n!)	Explodes
Exponential	$O(x^n)$	Explodes
Polynomial	$O(n^x)$	Explodes
Linear	O(n)	Proportional
Logarithmic	O(log n)	Levels Off
Constant	O(1)	Constant

Big O Comparison

Growth	Big O	Resource Usage
Factorial	O(n!)	Explodes
Exponential	$O(x^n)$	Explodes
Polynomial	$O(n^x)$	Explodes
Linear	O(n)	Proportional
Logarithmic	O(log n)	Levels Off
Constant	O(1)	Constant

Big O Comparison

Growth	Big O	Resource Usage
Factorial	$O(n!)$	Explodes
Exponential	$O(x^n)$	Explodes
Polynomial	$O(n^x)$	Explodes
Linear	$O(n)$	Proportional
Logarithmic	$O(\log n)$	Levels Off
Constant	$O(1)$	Constant

Big O Comparison

Growth	Big O	Resource Usage
Factorial	$O(n!)$	Explodes
Exponential	$O(x^n)$	Explodes
Polynomial	$O(n^x)$	Explodes
Linear	$O(n)$	Proportional
Logarithmic	$O(\log n)$	Levels Off
Constant	$O(1)$	Constant

Big O Comparison

Growth	Big O	Resource Usage
Factorial	$O(n!)$	Explodes
Exponential	$O(x^n)$	Explodes
Polynomial	$O(n^x)$	Explodes
Linear	$O(n)$	Proportional
Logarithmic	$O(\log n)$	Levels Off
Constant	$O(1)$	Constant