CSE 220
Data Structures

Lecture 01:
Introduction to Time and Space Complexity

Anwarul Bashir Shuaib [ABS]
Lecturer
Department of Computer Science and Engineering
BRAC University

Topics to Review

- 1D Arrays:
  - Declaration, initialization, and traversal.
  - Insertion, deletion, and updating elements.
  - Searching (Linear Search).
  - Sorting (Bubble Sort, Selection Sort, Insertion Sort).
- 2D Arrays:
  - Declaration, initialization, and traversal.
  - Row-major and column-major storage.
  - Common operations (matrix addition, multiplication, transpose).
- Pointers (Optional but Useful) – Understanding array-pointer relationship for dynamic memory allocation.

Why Is Complexity Important?

- Efficiency Matters:
  - As input size grows, some algorithms become too slow or use too much memory.

Why Is Complexity Important?

- Efficiency Matters:
  - As input size grows, some algorithms become too slow or use too much memory.
  - Complexity helps us predict how an algorithm will perform as input scales.

Why Is Complexity Important?

- Efficiency Matters:
  - As input size grows, some algorithms become too slow or use too much memory.
  - Complexity helps us predict how an algorithm will perform as input scales.
- Real-World Problems:
  - Computers are fast, but real-world problems can be huge (e.g., multiplying giant matrices or searching through millions of records).

Why Is Complexity Important?

- Efficiency Matters:
  - As input size grows, some algorithms become too slow or use too much memory.
  - Complexity helps us predict how an algorithm will perform as input scales.
- Real-World Problems:
  - Computers are fast, but real-world problems can be huge (e.g., multiplying giant matrices or searching through millions of records).
  - Even with powerful computers, inefficient programs can take too long or use too much memory.

Which Code Runs Faster?

```
def find_sum(arr):
    total = 0
    for num in arr:
        total += num
    return total
```

```
def find_pairs(arr):
    pairs = []
    for i in arr:
        for j in arr:
            pairs.append((i, j))
    return pairs
```

Which Code Runs Faster?

```python
def find_sum(arr):
    total = 0
    for num in arr:
        total += num
    return total
```

$$O(n)$$

```python
def find_pairs(arr):
    pairs = []
    for i in arr:
        for j in arr:
            pairs.append((i, j))
    return pairs
```

$$O(n^2)$$

What is Big-O? (Simplified!)

- Big-O is like a "*speedometer*" for your code.

What is Big-O? (Simplified!)

- Big-O is like a "*speedometer*" for your code.
  - It tells you **how fast your code runs** or **how much memory it uses** as the input grows.

What is Big-O? (Simplified!)

- Big-O is like a "*speedometer*" for your code.
  - It tells you **how fast your code runs** or **how much memory it uses** as the input grows.
  - **It’s Not Exact:** Big-O doesn’t count every single step. Instead, it gives you a general idea of **how your code scales.**

What is Big-O? (Simplified!)

- Big-O is like a "*speedometer*" for your code.
  - It tells you **how fast your code runs** or **how much memory it uses** as the input grows.
  - **It's Not Exact:** Big-O doesn't count every single step. Instead, it gives you a general idea of **how your code scales.**
- Example:
  - If your code takes $3n^2 + 2n + 1$ steps, Big-O simplifies it to $O(n^2)$

What is Big-O? (Simplified!)

- Big-O is like a "*speedometer*" for your code.
  - It tells you **how fast your code runs** or **how much memory it uses** as the input grows.
  - **It's Not Exact:** Big-O doesn't count every single step. Instead, it gives you a general idea of **how your code scales.**
- Example:
  - If your code takes $3n^2 + 2n + 1$ steps, Big-O simplifies it to $O(n^2)$
  - Why? Because as n gets really big, the n^2 part dominates the others

How do we find the complexity?

```
def add_nums(a,b,c):
    sum = a + b + c    # Constant time
    return sum         # Constant time
```

How do we find the complexity?

```
def add_nums(a,b,c):
    sum = a + b + c   # Constant time
    return sum        # Constant time
```

Exact run time complexity: $c + c = 2c$

How do we find the complexity?

```python
def find_sum(arr):
    total = 0          # Constant time
    for num in arr:    # Loop runs 'n' times
        total += num   # Constant time
    return total       # Constant time
```

Exact run time complexity: $c + nc + c = c(n+2)$

How do we find the complexity?

```python
def find_element_and_sum(arr, target):
    element = None       # Constant time
    sum = 0              # Constant time
    for i in arr:        # O(n)
        if i == target: # Constant time
            element = i # Constant time

    for i in arr:        # O(n)
        sum += i         # Constant time

    return element, sum # Constant time
```

Exact run time complexity: $c + c + n \cdot (c + c) + nc + c = 3c(n + 1)$

How do we find the complexity?

```python
def find_element_and_sum(arr, target):
    element = None       # Constant time
    sum = 0              # Constant time
    for i in arr:        # O(n)
        if i == target:  # Constant time
            element = i  # Constant time

    for i in arr:        # O(n)
        sum += i         # Constant time

    return element, sum  # Constant time
```

We only consider the worst-case scenario

Exact run time complexity: $c + c + n \cdot (c + c) + nc + c = 3c(n + 1)$

How do we find the complexity?

```python
def find_pairs(arr):
    pairs = []                       # Constant time
    for i in arr:                    # O(n)
        for j in arr:                # O(n) --> Nested!
            pairs.append((i, j))     # Constant time
    return pairs                     # Constant time
```

Exact run time complexity: $c + n \cdot n \cdot c + c = c(n^2 + 2)$

How do we find the complexity?

```python
def find_pairs(arr):
    pairs = []                      # Constant time
    for i in arr:                   # O(n)
        for j in arr:               # O(n) --> Nested!
            pairs.append((i, j))    # Constant time
    return pairs                    # Constant time
```

Exact run time complexity: $c + n \cdot n \cdot c + c = c(n^2 + 2)$

Think: Why n*n instead of n+n?

What Is ***Asymptotic*** Complexity?

- Big-O focuses on how an algorithm’s performance scales as the input size (n) grows.

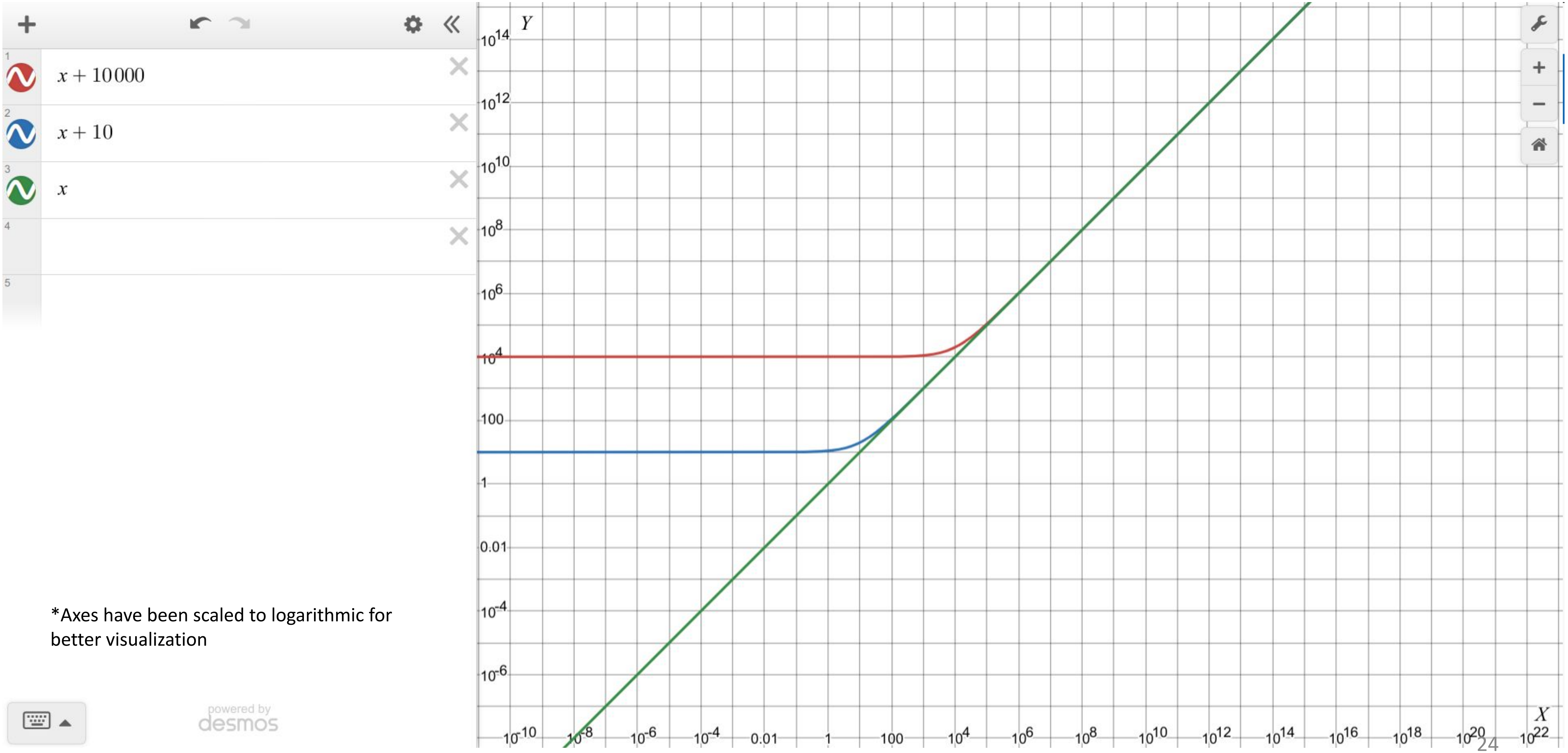
What Is ***Asymptotic*** Complexity?

- Big-O focuses on how an algorithm’s performance scales as the input size (n) grows.
- We **ignore constant factors and lower-order terms** because they become insignificant for large inputs.

What Is ***Asymptotic*** Complexity?

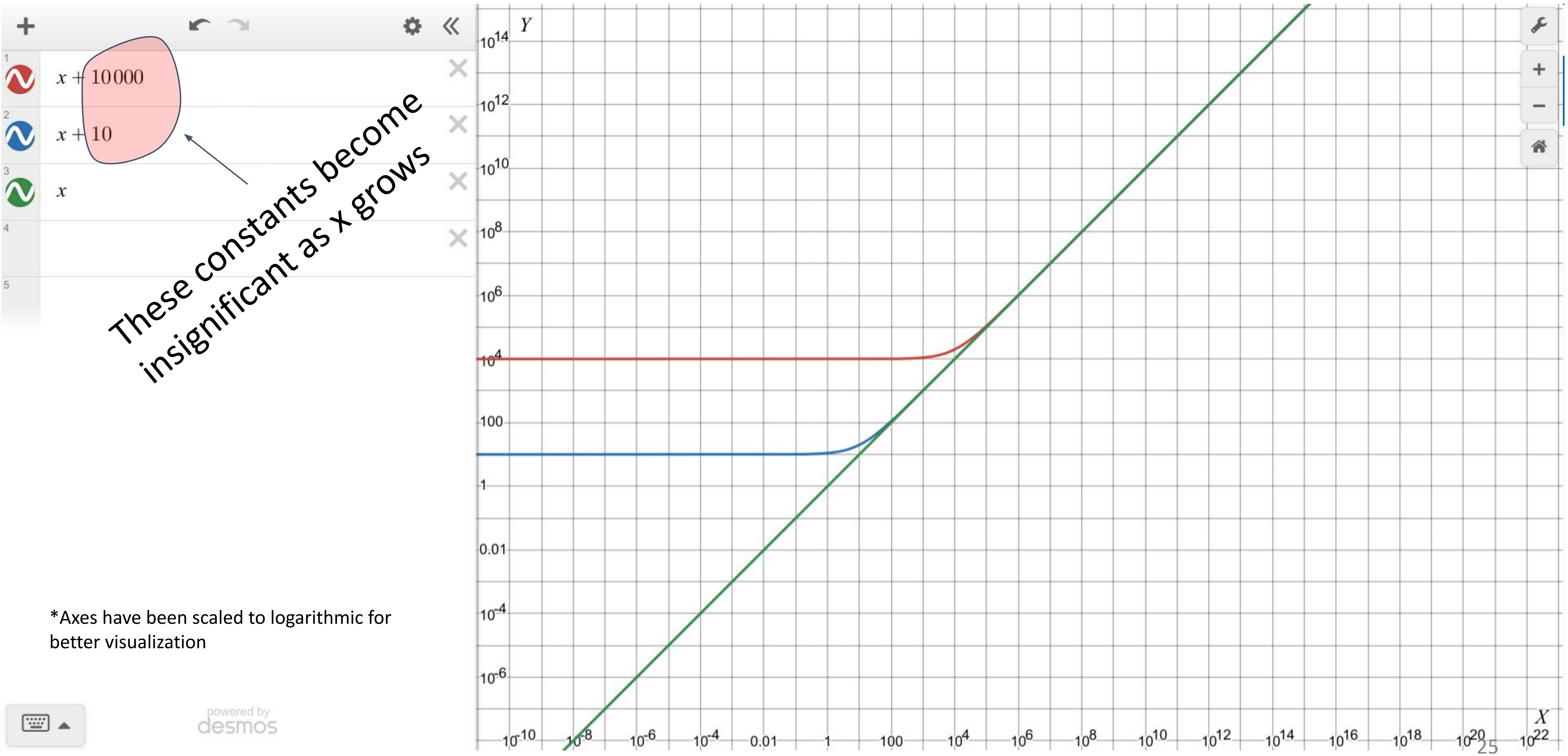
- Big-O focuses on how an algorithm's performance scales as the input size (n) grows.
- We **ignore constant factors and lower-order terms** because they become insignificant for large inputs.
- This Big-O notation is sometimes called the **"Order of Growth"**

What Is Asymptotic Complexity?



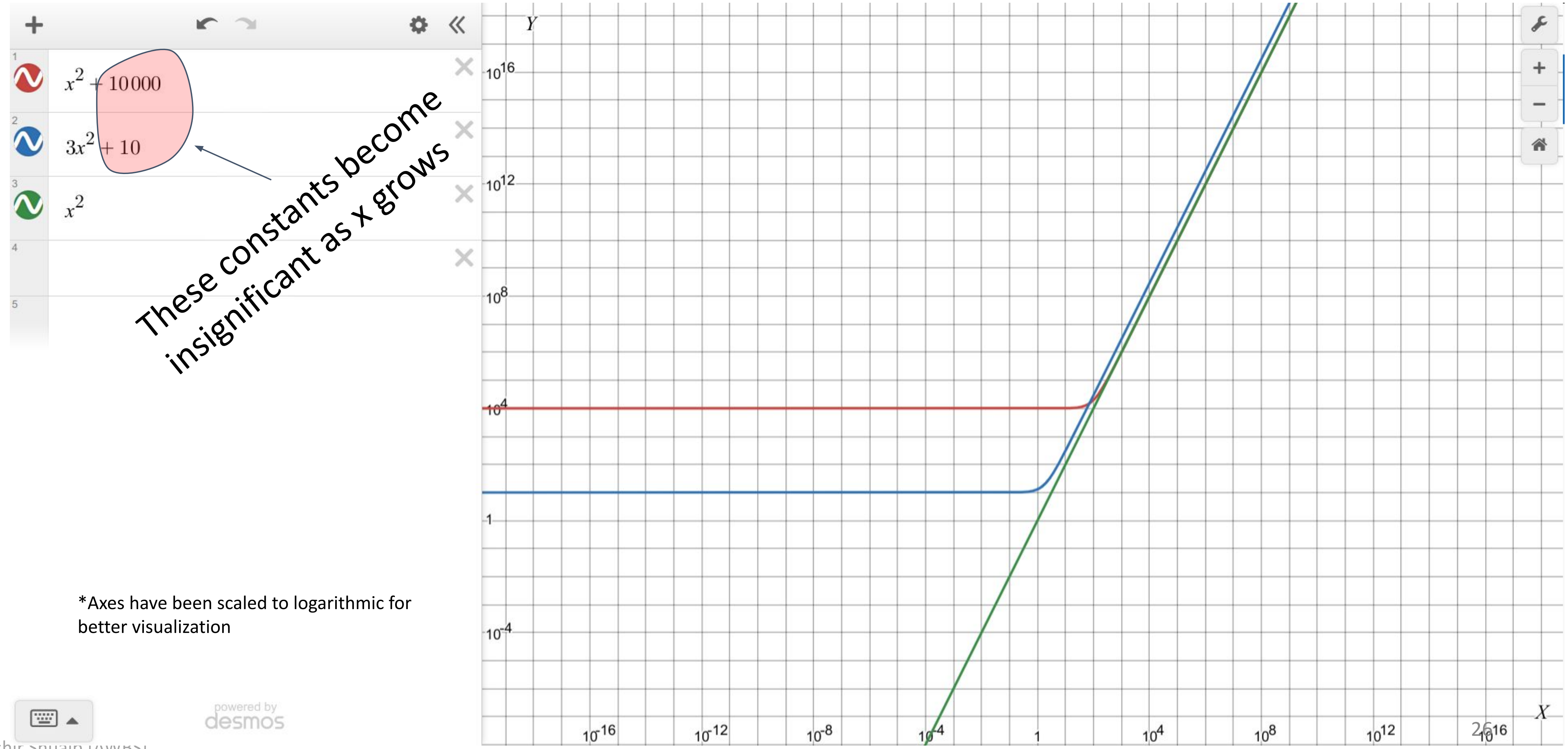
*Axes have been scaled to logarithmic for better visualization

What Is Asymptotic Complexity?



*Axes have been scaled to logarithmic for better visualization

What Is Asymptotic Complexity?



*Axes have been scaled to logarithmic for better visualization

How do we find the order of growth?

```python
def add_nums(a,b,c):
    sum = a + b + c    # Constant time
    return sum         # Constant time
```

Exact run time complexity: $c + c = 2c$

How do we find the order of growth?

```
def add_nums(a,b,c):
    sum = a + b + c    # Constant time
    return sum         # Constant time
```

Exact run time complexity: $c + c = 2c$

Asymptotic complexity: $O(1)$

How do we find the order of growth?

```python
def add_nums(a,b,c):
    sum = a + b + c    # Constant time
    return sum         # Constant time
```

Exact run time complexity: $c + c = 2c$

Asymptotic complexity: $O(1)$ $\rightarrow$ Constant Time

How do we find the order of growth?

```python
def find_sum(arr):
    total = 0          # Constant time
    for num in arr:    # Loop runs 'n' times
        total += num   # Constant time
    return total       # Constant time
```

Exact run time complexity: $c + nc + c = c(n + 2)$

How do we find the order of growth?

```python
def find_sum(arr):
    total = 0           # Constant time
    for num in arr:     # Loop runs 'n' times
        total += num    # Constant time
    return total        # Constant time
```

Exact run time complexity: $c + nc + c = c(n+2)$

Asymptotic complexity: $O(n)$ $\longrightarrow$ Linear Time

How do we find the order of growth?

```python
def find_element_and_sum(arr, target):
    element = None       # Constant time
    sum = 0              # Constant time
    for i in arr:        # O(n)
        if i == target: # Constant time
            element = i # Constant time

    for i in arr:        # O(n)
        sum += i         # Constant time

    return element, sum # Constant time
```

Exact run time complexity: $c + c + n \cdot (c + c) + nc + c = 3c(n + 1)$

How do we find the order of growth?

```python
def find_element_and_sum(arr, target):
    element = None       # Constant time
    sum = 0              # Constant time
    for i in arr:        # O(n)
        if i == target: # Constant time
            element = i # Constant time

    for i in arr:        # O(n)
        sum += i         # Constant time

    return element, sum # Constant time
```

Exact run time complexity: $c + c + n \cdot (c + c) + nc + c = 3c(n + 1)$

Asymptotic complexity: $O(n)$ $\longrightarrow$ Linear Time

How do we find the order of growth?

```python
def find_pairs(arr):
    pairs = []                       # Constant time
    for i in arr:                    # O(n)
        for j in arr:                # O(n) --> Nested!
            pairs.append((i, j)) # Constant time
    return pairs                     # Constant time
```

Exact run time complexity: $c + n \cdot n \cdot c + c = c(n^2 + 2)$

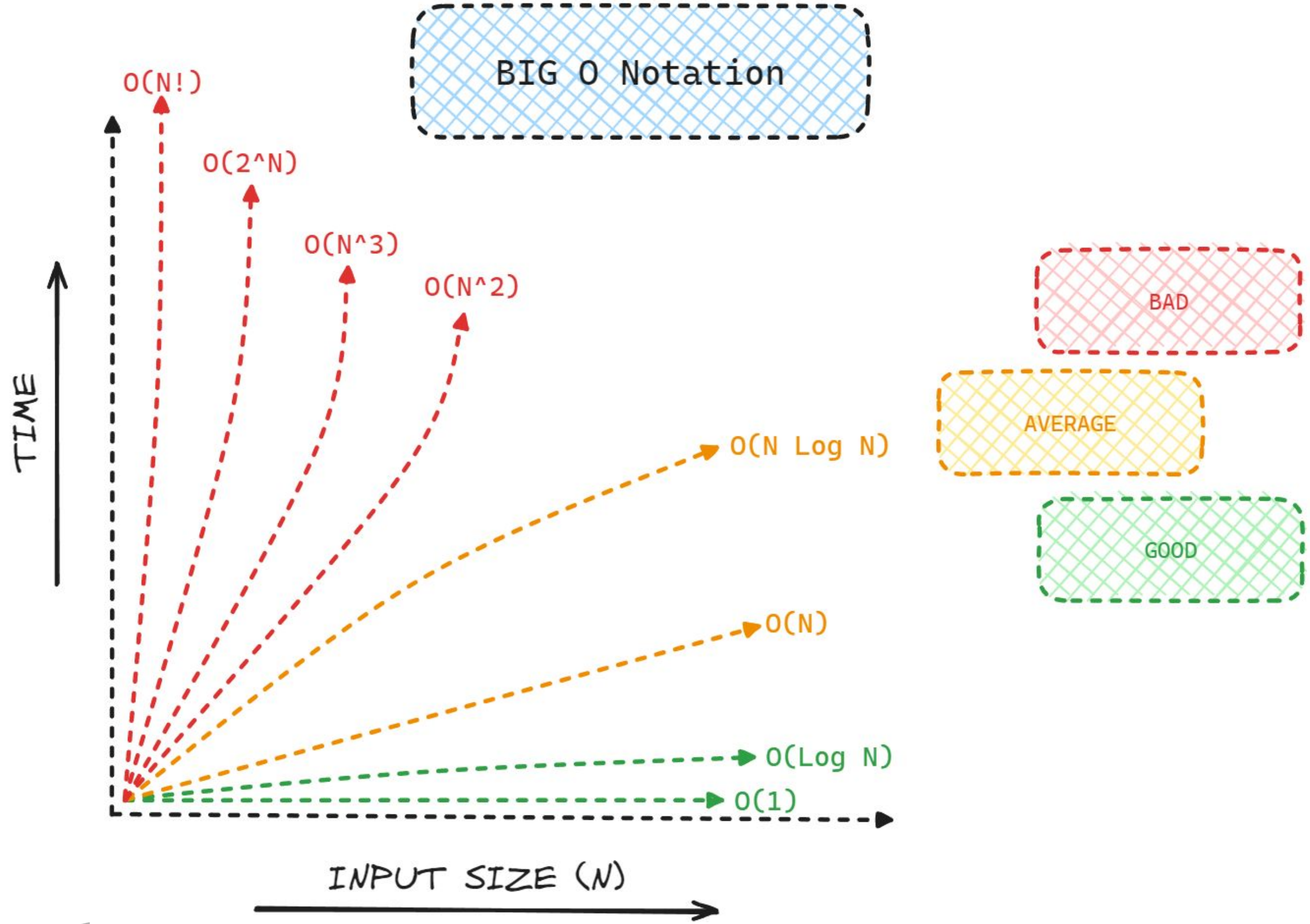
How do we find the order of growth?

```
def find_pairs(arr):
    pairs = []                       # Constant time
    for i in arr:                    # O(n)
        for j in arr:                # O(n) --> Nested!
            pairs.append((i, j))     # Constant time
    return pairs                     # Constant time
```

Exact run time complexity: $c + n \cdot n \cdot c + c = c(n^2 + 2)$

Asymptotic complexity: $O(n^2)$ $\rightarrow$ Quadratic Time

So...What Should We Prefer?



jwcarroll
@jwcarroll

Alternative Big O notation:

O(1) = O(yeah)
O(log n) = O(nice)
O(n) = O(ok)
O(n^2) = O(my)
O(2^n) = O(no)
O(n!) = O(mg!)

8:10 PM · 06 Apr 19 · Twitter for Android

Practice

- How would we perform a linear search in an array? What would be its time complexity (the order of growth)?

Practice

- How would we perform a linear search in an array? What would be its time complexity (the order of growth)?

```python
def linear_search(arr, target):
    for i in arr:
        if i == target:
            return i
    return None
```

Practice

- How would we perform a linear search in an array? What would be its time complexity (the order of growth)?

```
def linear_search(arr, target):
    for i in arr:
        if i == target:
            return i
    return None
```

Time complexity: $O(n)$

Practice

- How would we perform a linear search in an array? What would be its time complexity (the order of growth)?

```python
def linear_search(arr, target):
    for i in arr:
        if i == target:
            return i
    return None
```

Time complexity: $O(n)$

Think: Why O(n)? Why not O(1) when the target is at arr[0]?

Practice

- Can we improve it? What if the array is sorted beforehand?

Practice

- Can we improve it? What if the array is sorted beforehand?

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

See a nice demo here

Practice

- Can we improve it? What if the array is sorted beforehand?

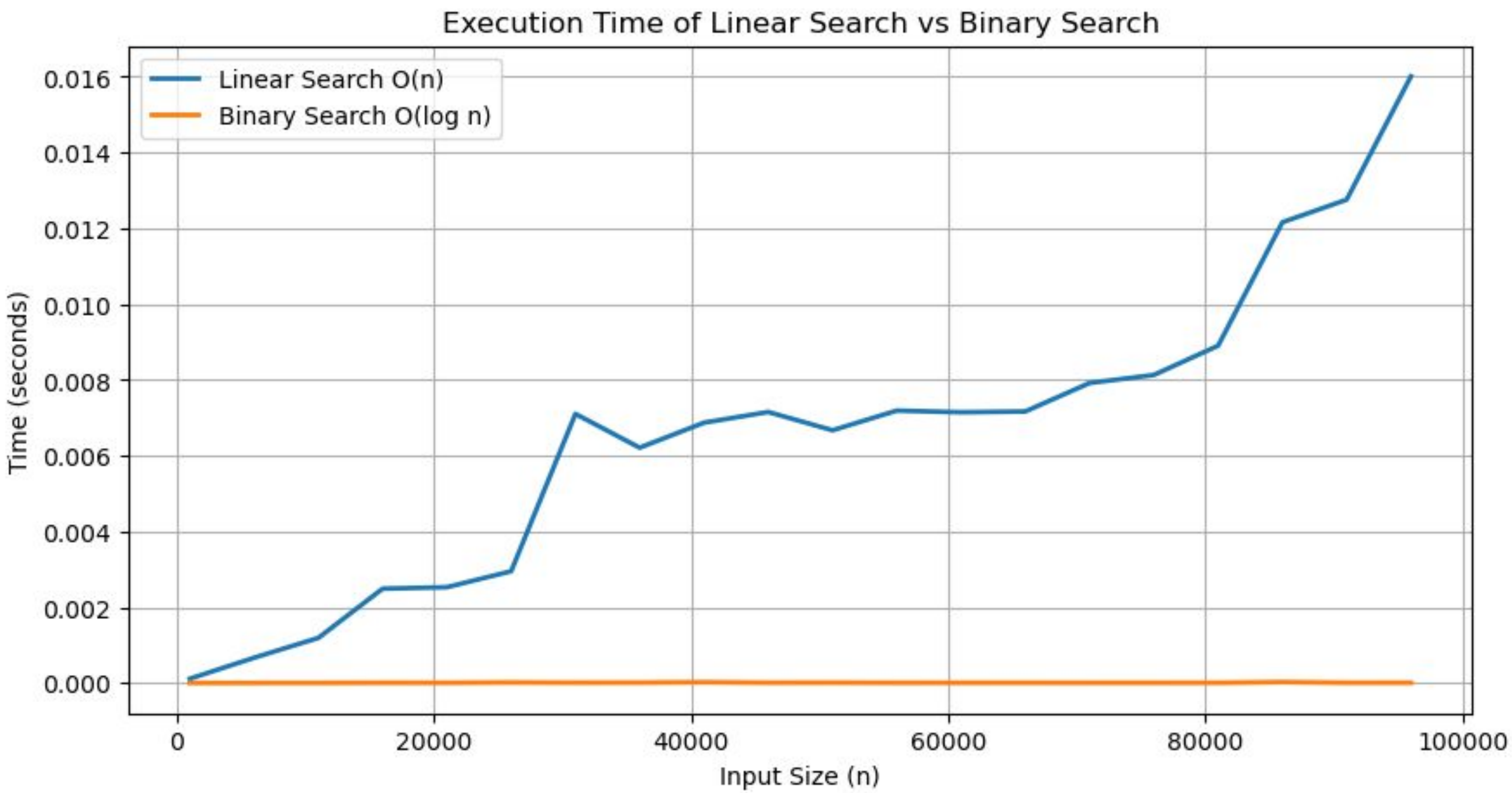
```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return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

See a nice demo [here]

Time complexity: $O(\log n)$ $\Rightarrow$ Logarithmic time

Seeing the Impact!

Seeing the Impact!



Practice

- What about 2D matrix multiplication?

```python
def mat_mul(matA, matB):
    # matA: m x n
    # matB: n x p
    m, n = len(matA), len(matA[0])
    r, p = len(matB), len(matB[0])
    if n != r:
        print("Dimension mismatch")
        return None

    result = [[0 for _ in range(p)] for _ in range(m)]
    for i in range(m):
        for j in range(p):
            for k in range(n):
                result[i][j] += matA[i][k] * matB[k][j]

    return result
```

Practice

- What about 2D matrix multiplication?

```python
def mat_mul(matA, matB):
    # matA: m x n
    # matB: n x p
    m, n = len(matA), len(matA[0])
    r, p = len(matB), len(matB[0])
    if n != r:
        print("Dimension mismatch")
        return None

    result = [[0 for _ in range(p)] for _ in range(m)]
    for i in range(m):
        for j in range(p):
            for k in range(n):
                result[i][j] += matA[i][k] * matB[k][j]

    return result
```

Time complexity: $O(n^3)$

The Power of Efficient Algorithms

- Converting an $O(n^3)$ algorithm to $O(n^2)$ can make the difference between impossible and practical.

The Power of Efficient Algorithms

- Converting an $O(n^3)$ algorithm to $O(n^2)$ can make the difference between impossible and practical.
- Say, n=1000000, 2GHz CPU (2x10^9 operations / second)
  - With O(n^3) ⇒
  - With O(n^2) ⇒

The Power of Efficient Algorithms

- Converting an $O(n^3)$ algorithm to $O(n^2)$ can make the difference between impossible and practical.
- Say, n=1000000, 2GHz CPU (2x10^9 operations / second)
  - With O(n^3) ⇒ 10^18 operations
  - With O(n^2) ⇒ 10^12 operations

The Power of Efficient Algorithms

- Converting an $O(n^3)$ algorithm to $O(n^2)$ can make the difference between impossible and practical.
- Say, n=1000000, 2GHz CPU (2x10^9 operations / second)
  - With O(n^3) ⇒ 10^18 operations ⇒ approx 16 years!
  - With O(n^2) ⇒ 10^12 operations ⇒ 8.3 minutes!

Space Complexity

- Same type of analysis goes for memory consumption as well

Space Complexity

- Same type of analysis goes for memory consumption as well
  - Space complexity measures how much memory an algorithm uses as the input size (n) grows.

Space Complexity

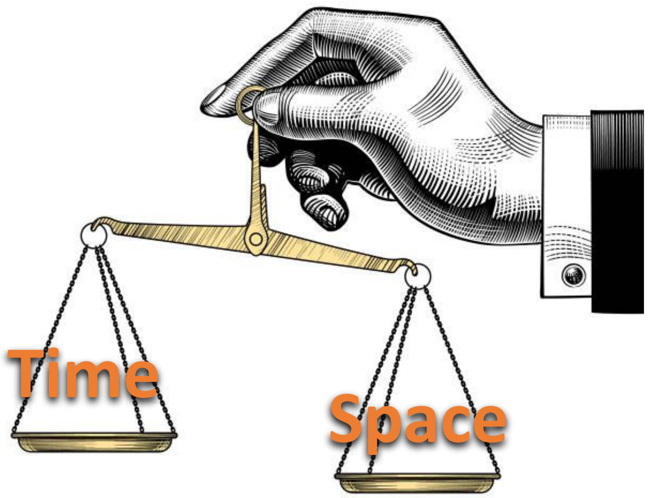
- Same type of analysis goes for memory consumption as well
  - Space complexity measures how much memory an algorithm uses as the input size (n) grows.
  - Memory is valuable!
  - 1D array
  - 2D array

Space Complexity

- Same type of analysis goes for memory consumption as well
  - Space complexity measures how much memory an algorithm uses as the input size (n) grows.
  - Memory is valuable!
  - 1D array ⇒ O(n) space complexity
  - 2D array ⇒ O(n^2) space complexity

Time Complexity vs Space Complexity

- **Time Complexity:**
  - Measures how fast an algorithm runs.
- **Space Complexity:**
  - Measures how much memory an algorithm uses.
- **Trade-Off:**
  - Sometimes, you can save time by using more memory, or save memory by using more time.



Examples

```
def sum_numbers(a, b):
    return a + b                         # O(1) space


def create_1d_array(size):               # O(n) space
    return [0] * size


def create_2d_array(rows, cols):         # O(n^2) space
    return [[0] * cols for _ in range(rows)]
```

Pitfall

String concatenation may not be constant as we might think!

```
def bad_string_concat(n):
    s = ""
    for i in range(n):
        s += "a"
    return s
```

Pitfall

String concatenation may not be constant as we might think!

```
def bad_string_concat(n):
    s = ""
    for i in range(n):
        s += "a"  # O(n) operation inside O(n) loop → O(n²) total
    return s
```

Some Common Order of Growth Functions

order of growth	name	typical code framework	description	example
1	**constant**	`a = b + c;`	statement	add two numbers
$\log N$	**logarithmic**	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search
N	**linear**	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort
N^2	**quadratic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs
N^3	**cubic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples
2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets

Things Can Get Complicated 😬

```python
def sum_array(arr, i = 0):
    if i == len(arr):
        return 0
    return arr[i] + sum_array(arr, i + 1)
```

O(n)

```python
def binary_search(arr, target, low = 0, high = None):
    if high is None:
        high = len(arr) - 1
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search(arr, target, mid + 1, high)
    else:
        return binary_search(arr, target, low, mid - 1)
```

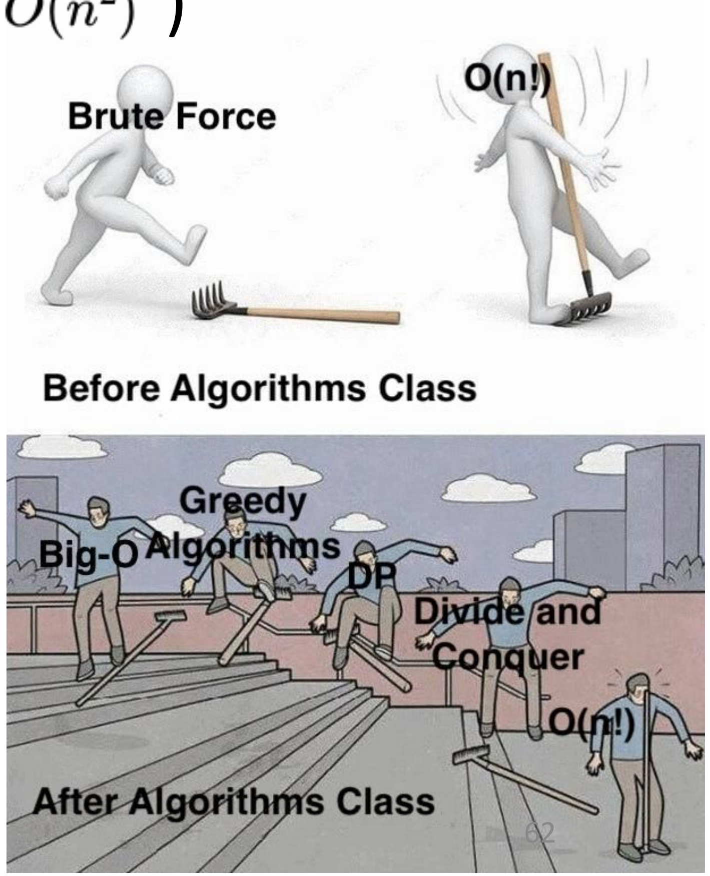
O(logn)

```python
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

O(2^n)

Conclusion: Efficiency Beats Raw Power

- A supercomputer can perform trillions of calculations per second.
- But if it runs an inefficient algorithm (e.g., $O(n^3)$ or $O(n^2)$) it can still struggle with large inputs.
- A normal computer running an efficient algorithm (e.g., $O(n)$ or $O(n \log n)$) can outperform a supercomputer running an inefficient one.



Homework

- You have an n-bit secret integer. What would be the time complexity of an algorithm that tries to guess this integer?
- Find the space complexity of storing all possible permutations of a given string.
- Find the time complexity of generating all possible subsets of a given set.
  - Hint: {1,2,3} ⇒ { {}, {1}, {2}, {3}, {1,2}, {2,3}, {3,1}, {1,2,3} }

Homework

- You have a list of integers. Find pairs of integers (a,b) such that a+b = 0
  - O(n^2) time ⇒ Brute force
  - O(nlogn) time ⇒ Sorting + Binary search
  - O(n) time ⇒ Hashmap (Will study later in this course)
  - Try to figure out the space complexity for each approach!

Reading Materials

- https://medium.com/@hlfdev/algorithms-discover-the-power-of-big-o-notation-17a367bd62a


Does he bite?
No, but he can hurt you in other ways.
Understanding the programming memes doesn't make you a dev