A Possible Problem (Interview Question)

You got a ***list of items****, where every item has a* ***value and a weight****. You got a bag that holds a* ***maximum weight of X****.*

Write a program that ***maximizes the value*** *of the items you put into the bag whilst ensuring that you* ***don't exceed the maximum weight****.*

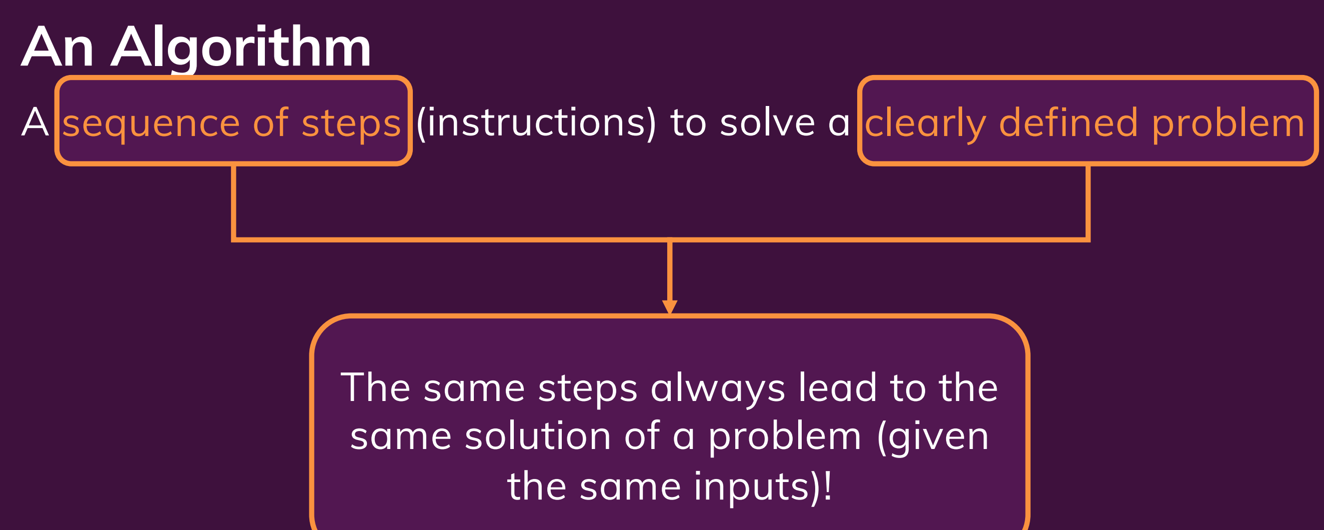
```
items = [
  {id: 'a', val: 10, w: 3},
  {id: 'b', val: 6, w: 8},
  {id: 'c', val: 3, w: 3}
]

maxWeight = 8

bag = ['a', 'c'] // solution
```

Knapsack problem

Value: 13
Weight: 6 (< 8)

This is being asked to check your problem-solving skills.

Algorithms: What and Why?



Every program is an algorithm! Or: Every program consists of many smaller algorithms

As a programmer, you **need to be able to solve problems** (efficiently)!

What is the "Best Possible Solution"?

Minimum amount of code?

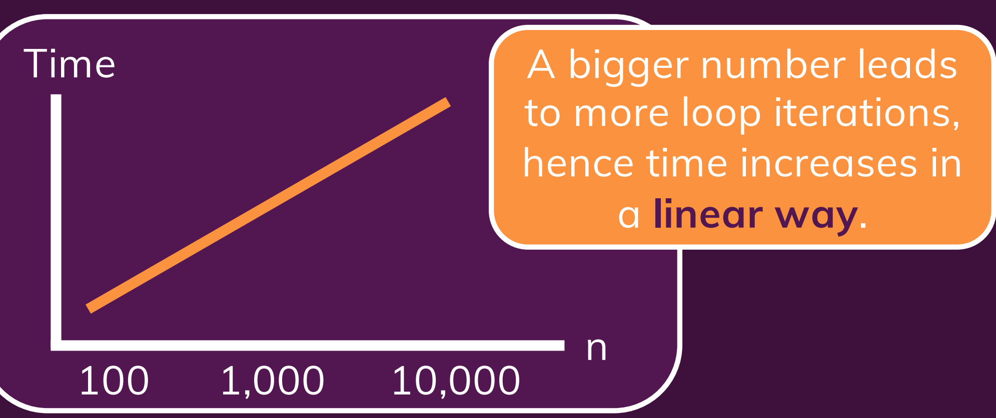
Best performance?

Least memory usage?

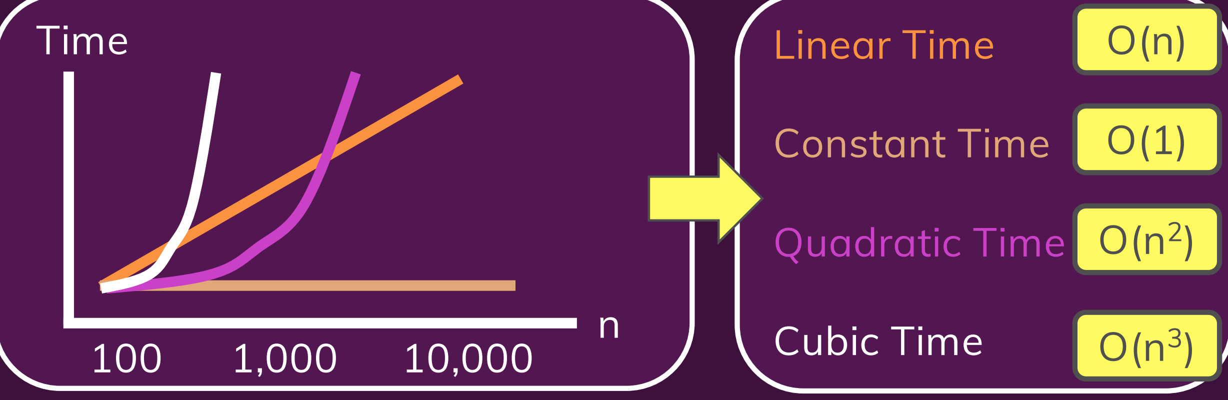
Personal preference?

Measuring Performance (Time Complexity – Big O)

```
function sumUp(n) {
  let result = 0;
  for (let i = 1; i <= n; i++) {
    result += i;
  }
  return result;
}
```



A bigger number leads to more loop iterations, hence time increases in a **linear way**.



Linear Time	$O(n)$
Constant Time	$O(1)$
Quadratic Time	$O(n^2)$
Cubic Time	$O(n^3)$

We care about the **trend/ kind of function**.

Big O Notation

Deriving the Time Complexity Function

n = 1 | n = 3 | n = 10 | n = n

```
function sumUp(n) {
  let result = 0;
  for (let i = 1; i <= n; i++) {
    result += i;
  }
  return result;
}
```

Count the number of expression executions.

Deriving the Time Complexity Function

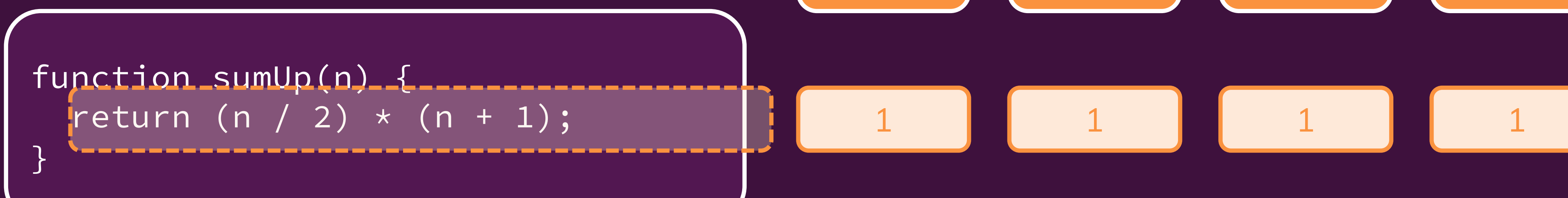
```
function sumUp(n) {
  let result = 0;
  for (let i = 1; i <= n; i++) {
    result += i;
  }
  return result;
}
```

Expression	n = 1	n = 3	n = 10	n = n
let result = 0;	1	1	1	1
for (let i = 1; i <= n; i++) {	1	1	1	1
result += i;	1	3	10	n
return result;	1	1	1	1

Count the number of expression executions.

T =

Deriving Constant Time Complexity




```
function sumUp(n) {
  return (n / 2) * (n + 1);
}
```

Count the number of expression executions.

T = 1

Deriving Big O (Asymptotic Analysis)

Step	Action	Example
1	Define the function	T = 1*n + 3
2	Find the **fastest growing term**	T = **a*n** + b
3	Remove the coefficient	T = ~~1~~*n

T = n

O(n)

Deriving Big O (Asymptotic Analysis)

Step	Description	Result
1	Define the function	T = 1
2	Find the **fastest growing term**	T = 1
3	Remove the coefficient	T = 1

T = 1

O(1)

Using Big O to Compare Algorithms

O(1)	Constant Time Complexity	n (number of input) has **no effect** on the time the algorithm takes
O(log n)	Logarithmic Time Complexity	Execution time grows **logarithmically** with n
O(n)	Linear Time Complexity	Execution time grows **linearly** with n
$O(n^2)$	Quadratic Time Complexity	Execution time grows **quadratically** with n
$O(2^n)$	Exponential Time Complexity	Execution time grows **exponentially** with n

ACADEMIND

Practice Time!

Write an algorithm that takes an **array of numbers** as input and **calculates the sum** of those numbers.

Define the Time Complexity of that algorithm and determine what the **lowest possible Time Complexity** is for this problem.

```
function sumNumbers(numbers) { ??? }

sumNumbers([1, 3, 10]) // should yield 14
```

Your task!

About this Course

What & Why

Examples & Different Algorithms

Different Solution Approaches: Recursion, Dynamic Programming, Greedy Algorithms

A Solid Foundation & Plan

Course Outline

- Basics & Time Complexity
- Math Algorithms
- Recursion & Dynamic Programming
- Search Algorithms
- Sorting Algorithms
- Space Complexity
- Sets (Arrays) Algorithms
- More Complex Algorithms & A “Blueprint”