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?

An Algorithm

A sequence of steps (instructions) to solve a clearly defined problem

The same steps always lead to the same solution of a problem (given the same inputs)!

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)!

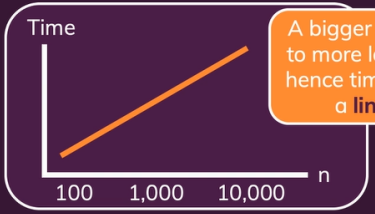
ACADEMIND

What is the “Best Possible Solution”?

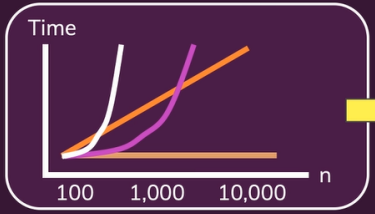
- Minimum amount of code?
- Best performance?
- Least memory usage?
- Personal preference?

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

ACADEMIND

Deriving the Time Complexity Function

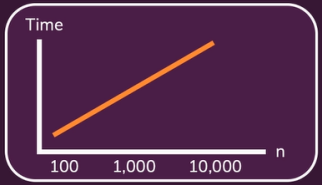
	n = 1	n = 3	n = 10	n = n
`let result = 0;`	1	1	1	1
`for (let i = 1; i <= n; i++) {`	1	1	1	1
`result += i;`	1	3	10	n
`return result;`	1	1	1	1

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

Count the number of expression executions.

T = 1 + 1 + n + 1 = 3 + n = **3 + 1 * n**

Deriving Big O (Asymptotic Analysis)



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

T = n

O(n)

Deriving Constant Time Complexity

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

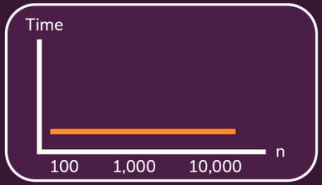
```
function sumUp(n) {
  return (n / 2) * (n + 1);
}
```

Count the number of expression executions.

T = 1

ACADEMIND

Deriving Big O (Asymptotic Analysis)



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

T = 1

O(1)

ACADEMIND

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!

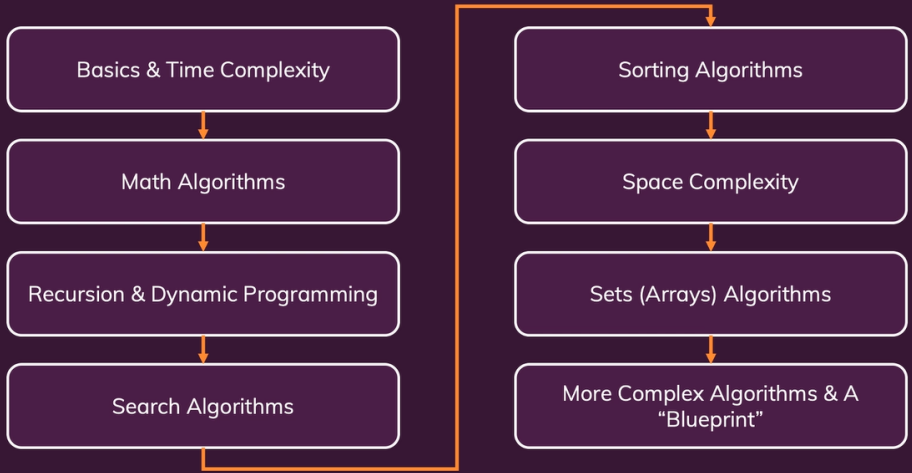
ACADEMIND

About this Course

- What & Why
- Examples & Different Algorithms
- Different Solution Approaches: Recursion, Dynamic Programming, Greedy Algorithms
- A Solid Foundation & Plan

ACADEMIND

Course Outline



ACADEMIND

Math Algorithms

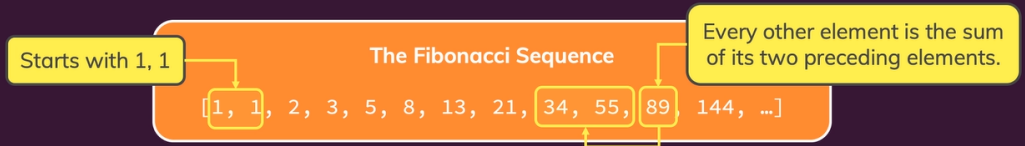
Fun With Numbers

ACADEMIND

Module Content

- Explore Math Algorithms
- Get a Better Feeling for Big O Notation
- How to Solve Problems

Fibonacci Sequence



Popular Interview Question

Return the nth element of the Fibonacci sequence.
e.g. `fib(4)` yields 5

Algorithm needs to do **two things**:

(1) Calculate the sequence up to the element we're looking for

(2) Return that element

Primality Test

Determine whether an input number is a prime number

```
isPrime(9); // false
isPrime(5); // true
```

Algorithm needs to do **one thing**:

(1) Try dividing the number by all smaller numbers and return true if it's only divisible by itself and 1

Big O – Best Case, Worst Case, Average Case

For some problems (e.g. sorting array items), you have **different cases** with different time complexities

Example: "Sort the numbers in array from small to big"
Algorithm used: "Bubble Sort" *(covered later)*

Best Case: Already sorted `nums = [1, 2, 3]`	**Avg. Case**: Random order `nums = [2, 3, 1]`	**Worst Case**: Inverse order `nums = [3, 2, 1]`
$O(n)$	$O(n^2)$	$O(n^2)$

ACADEMIND

Primality Test - Improved!

Two Examples:

5 is a prime: Divisible by itself and by 1
9 is NOT a prime: Divisible by itself, by 1 and by 3

Every number, that's NOT a prime has a product that consists of **two factors a & b** that are both **neither 1 nor the number** itself.

9 = 3 * 3

At least one factor is smaller or equal to the square root of the number.

sqrt(9) = 3 ➔ 9 = 3 * 3 ➔ Both factors are equal to the square root

ACADEMIND

Practice Time!

Write **two algorithms**:
(1) The first algorithm should take **an array of numbers** as input and **return the minimum value** in the array (i.e. the smallest number)
(2) The second (**independent**) algorithm should take **a number as input** and return **"true" if it's an even number**, "false" for odd numbers

Also define the **time complexities** and **possible cases** for both algorithms!

ACADEMIND

Is Power Of Two

Determine whether an input number is a power of two

```
isPowerOfTwo(8); // true
isPowerOfTwo(5); // false
```

Algorithm needs to do **one thing**:

(1) Divide number and future division results by two, until 1 is reached and check if the remainder is always 0

Is Power Of Two - Improved!

Bitwise Magic!

(Unsigned) Powers of two, in binary form, always have just one bit:

```
1: 1
2: 10
4: 100
```

A bitwise operation can be used to check if a number is power of two:

```
number & (number - 1)  === 0 // true: it's power of two
```

ACADEMIND

Determining the “Nature of an Algorithm”

O(n)	Higher n leads to a linear increase in runtime	Look for (single) loops
O(1)	Higher n does not affect runtime	Look for functions without loops and without any function calls
O(log n)	Runtime grows with n but at a much slower pace	Look for functions where n is split (divided) into smaller “chunks”

Factorial

Calculate the factorial of a number

```
fact(3); // 3 * 2 * 1 = 6
fact(5); // 5 * 4 * 3 * 2 * 1 = 120
```

Algorithm needs to do **one thing**:

(1) Go through all smaller numbers and multiply them with each other (and with the input number)

ACADEMIND

Recursion & Dynamic Programming

Beyond Iterations

ACADEMIND

Module Content

What is "Recursion"?

Recursion in Algorithms (and why it's not always better!)

Dynamic Programming

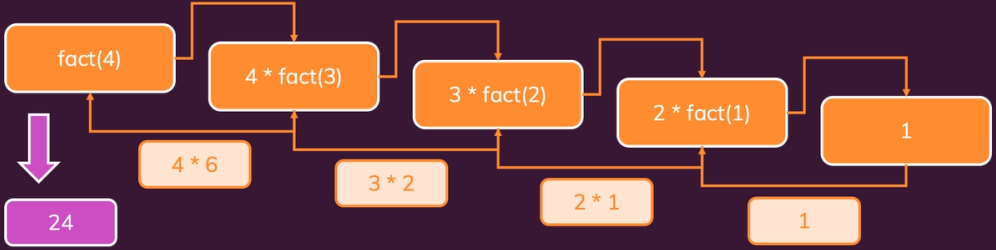
Factorial – Recursive Solution

Base Case

```
function fact(n) {
  if (n <= 1) {
    return 1;
  }

  return n * fact(n - 1);
}
```

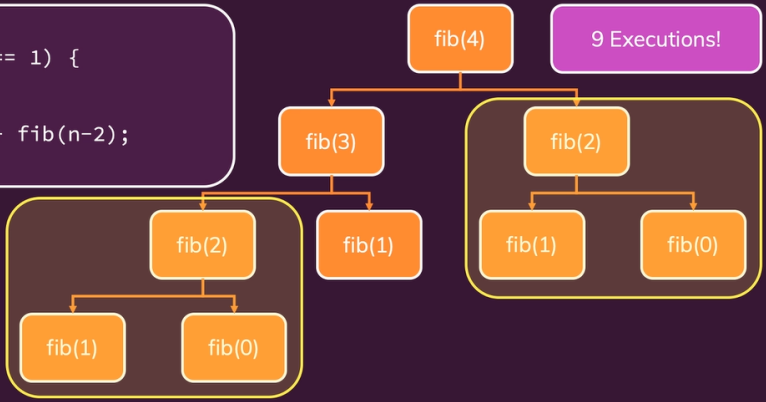
Recursive Step



Recursion Is Not Always Best

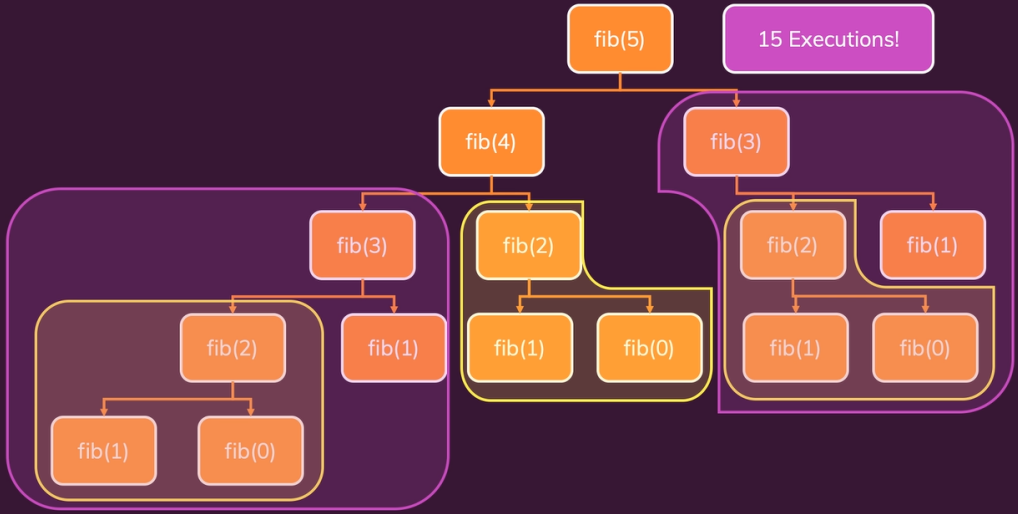
Recursive Fibonacci

```
function fib(n) {
  if (n == 0 || n == 1) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```



ACADEMIND

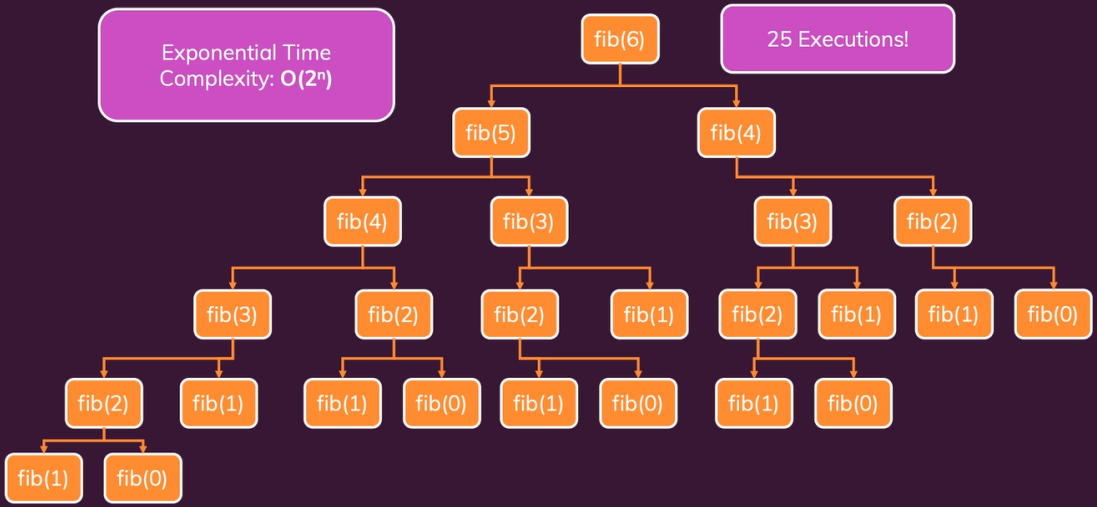
Recursion Is Not Always Best



ACADEMIND

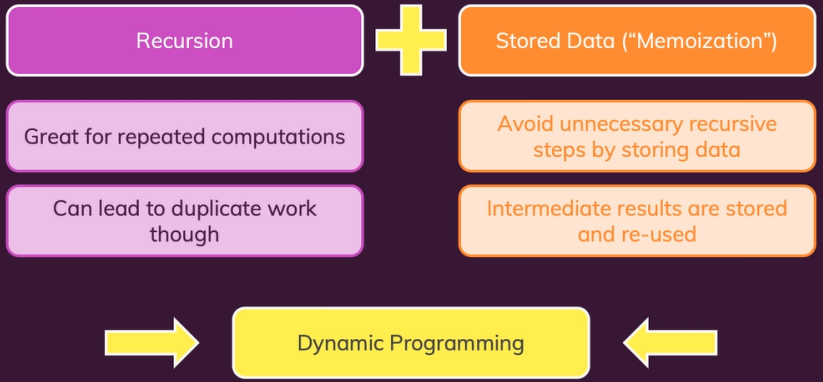
Recursion Is Not Always Best

Exponential Time Complexity: $O(2^n)$



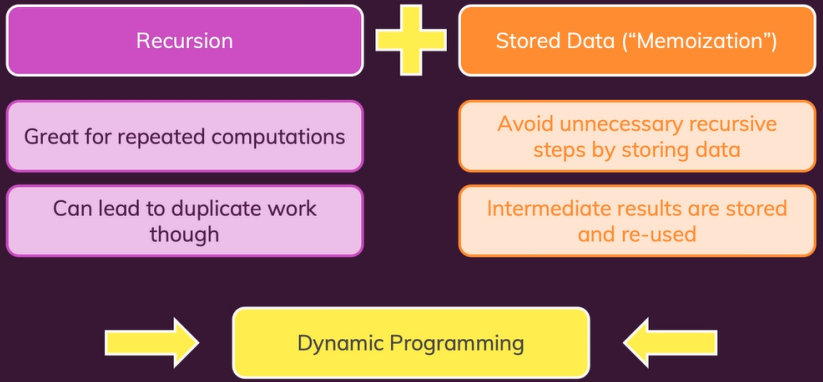
ACADEMIND

What is “Dynamic Programming”?



ACADEMIND

What is “Dynamic Programming”?



ACADEMIND

Or: Use the “Bottom-Up Approach”

Memo

Build it up “from the bottom”

```
function fib(n) {
  const numbers = [1, 1];
  for (let i = 2; i < n + 1; i++) {
    numbers.push(numbers[i - 2] + numbers[i - 1]);
  }
  return numbers[n];
}
```

ACADEMIND

Search Algorithms

Finding Things (Efficiently)

ACADEMIND

Module Content

What are “Search Algorithms” all about?

Linear Search

Binary Search & The “Master Theorem”

ACADEMIND

Search Algorithms?

Looking for 5

ACADEMIND

Search Algorithms?

You can use different algorithms for different kinds of lists – with different time complexities

ACADEMIND

Linear Search

Linear search works on **ordered and unordered** lists and **checks all items** until it finds the element you're searching.

ACADEMIND

Linear Search – Time Complexity

Best Case	Average Case	Worst Case
The item we're looking for is the very **first** item in the list	Random order, we don't know where the item is	The item we're looking for is the very **last** item in the list
O(1)	Tends to be O(n)	O(n)

ACADEMIND

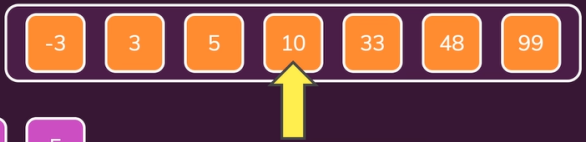
Binary Search

Looking for 5

Doesn't work on unordered lists! You need to sort the list first!

ACADEMIND

Binary Search



Looking for 5

Find median and compare it to the element you're trying to find

Is it the element you're looking for?

If element wasn't found, take the half in which must be inside

Repeat!

Binary Search – Time Complexity

Best Case	Average Case	Worst Case
The item we're looking for is **right in the middle**	We don't know where the item will be	The item we're looking for is **at the beginning or end**
O(1)	Tends to be O(log n)	O(log n)

Because we split the array in half in every iteration

Recursion & Big O (The Master Theorem)

How do you derive Big O for more complex recursive algorithms?

Master Theorem

Runtime of recursion: $O(n^{log_b a})$

Overall algorithm runtime (time complexity) - three cases:

Recursion does more work

$O(n^{log_b a})$

Same work inside and outside of recursion

$O(n^{log_b a} \log n)$

Non-recursive part does more work

$O(fn(n))$

where

a equals the number of subproblems (number of recursion splits)

b equals the relative subproblem size (input reduction per split)

f(n) equals the runtime outside of the recursion

ACADEMIND

Sorting Algorithms

Everything Needs To Be In Order

ACADEMIND

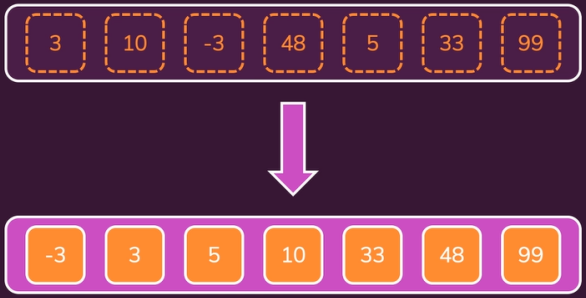
Module Content

What is “Sorting” all about?

Various Sorting Algorithms

ACADEMIND

What is “Sorting”?



Goal: Sort items – for example to then apply search algorithms

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

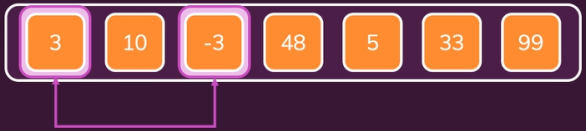
Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

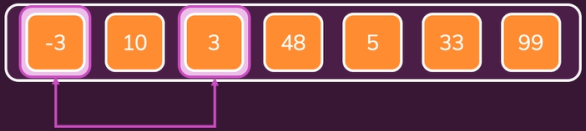
Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

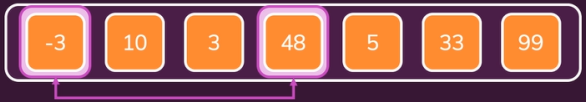
Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

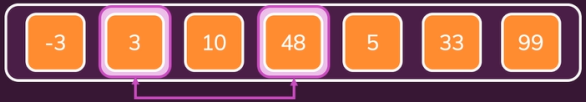
Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

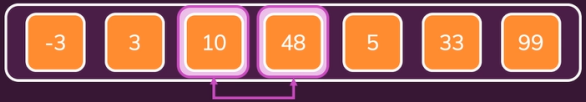
Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

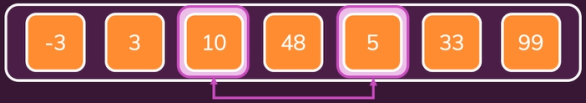
Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

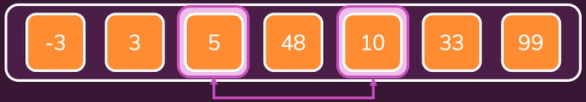
Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

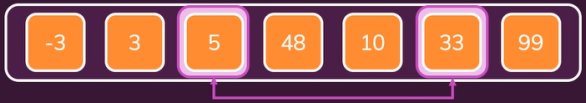
Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

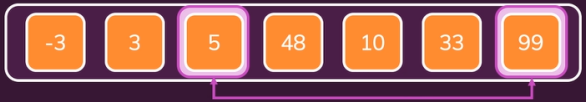
Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

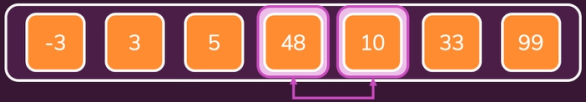
Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.



ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort

Compare two items at a time and sort them. Go **through the entire array multiple times** until all pairs were compared and sorted.

ACADEMIND

Bubble Sort – Time Complexity

Best Case	Average Case	Worst Case
Items are already sorted	Random order, we don't know where the item is	Items are sorted in wrong order
O(n)	Tends to be O(n^2)	O(n^2)

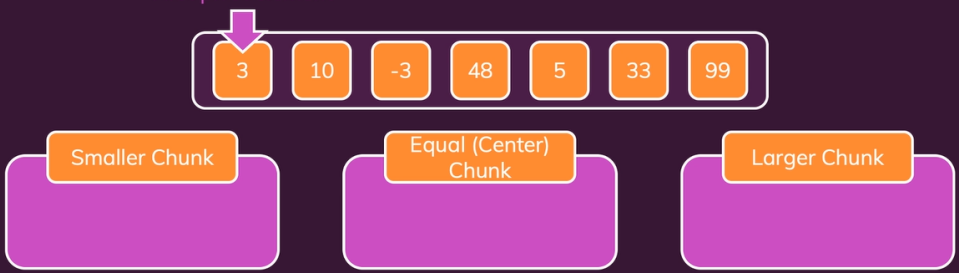
ACADEMIND

Quicksort

Use **pivot elements to split array** into **smaller chunks** – elements **bigger, smaller and equal than the pivot element**. **Repeat** that process for all chunks and concat the sorted chunks.

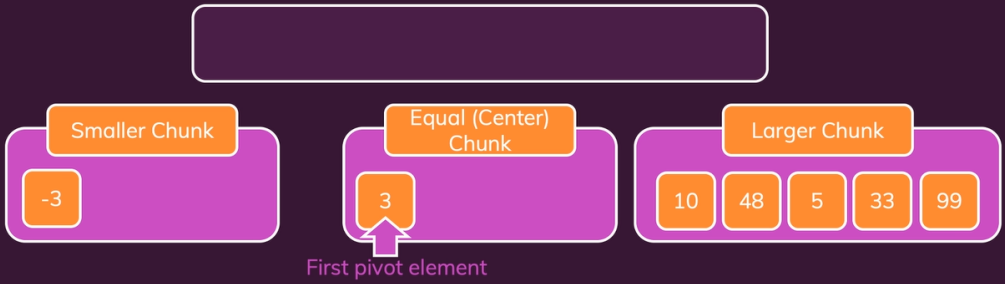
Quicksort

Use **pivot elements to split array** into **smaller chunks** – elements **bigger, smaller and equal than the pivot element**. **Repeat** that process for all chunks and concat the sorted chunks.



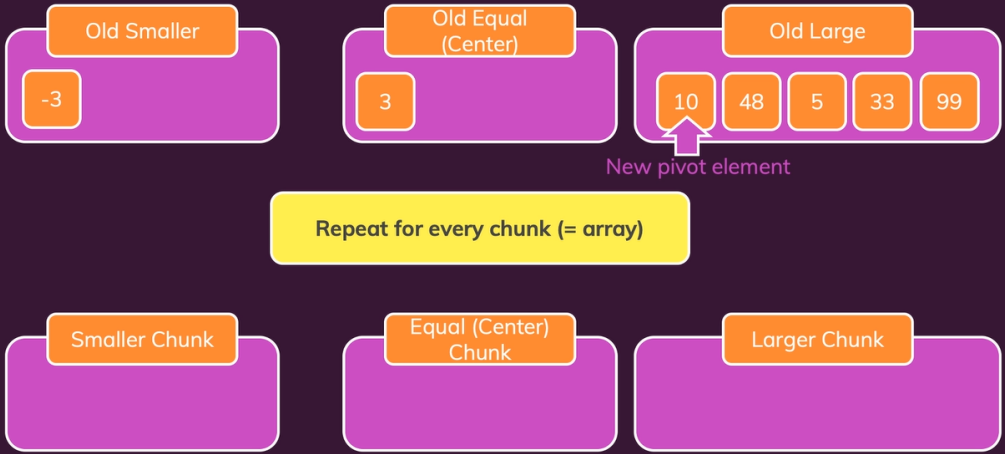
Quicksort

Use **pivot elements to split array** into **smaller chunks** – elements **bigger, smaller and equal than the pivot element**. **Repeat** that process for all chunks and concat the sorted chunks.

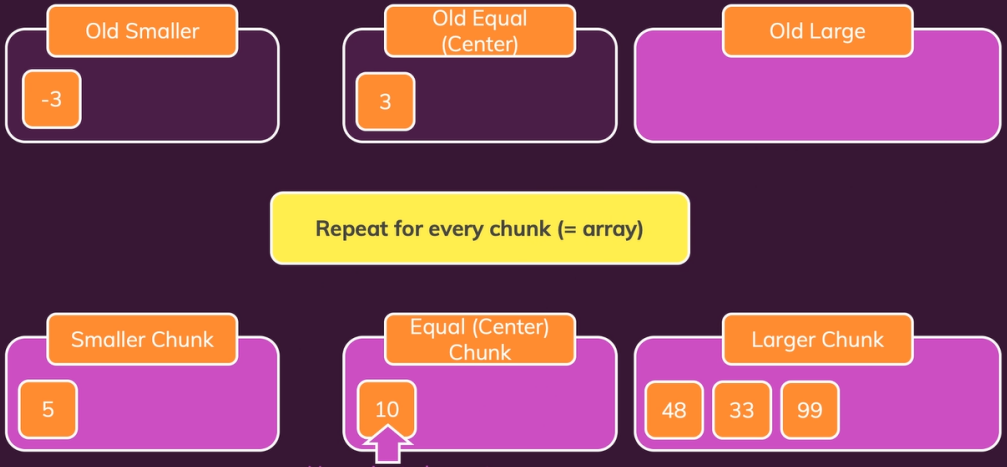


ACADEMIND

Quicksort



Quicksort


Old Smaller
-3
Old Equal (Center)
3
Old Large
Repeat for every chunk (= array)
Smaller Chunk
5
Equal (Center) Chunk
10
Larger Chunk
48
33
99
New pivot element

ACADEMIND

Quicksort

ACADEMIND

Quicksort

ACADEMIND

Quicksort – Time Complexity

Best Case	Average Case	Worst Case
Items are sorted randomly (NOT in right or wrong order)	Items are sorted randomly (NOT in right or wrong order)	Items are already sorted (order does not matter)
O(n * log n)	O(n * log n)	O(n^2)

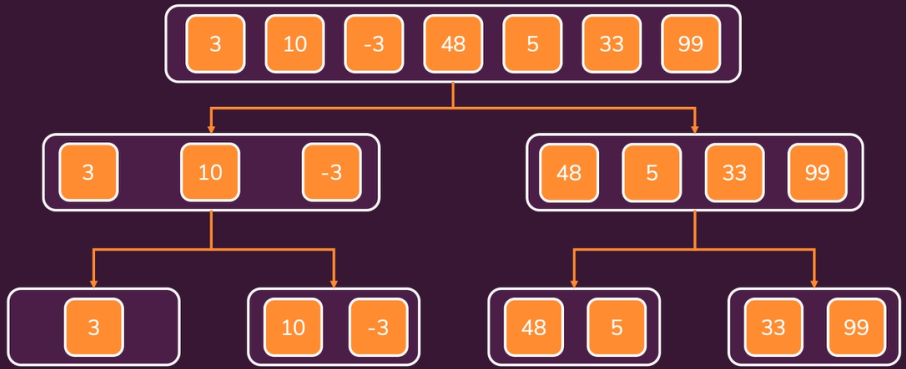
ACADEMIND

Merge Sort

Split array multiple times until we have only **2-element arrays** left – **sort those arrays** and **merge them back together**.

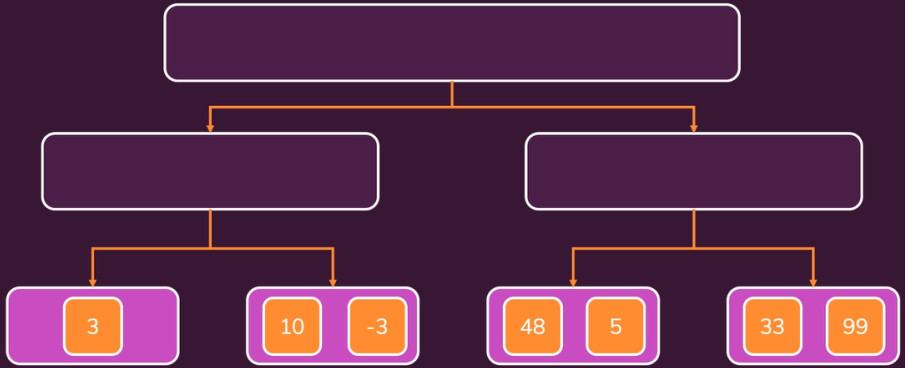
ACADEMIND

Merge Sort



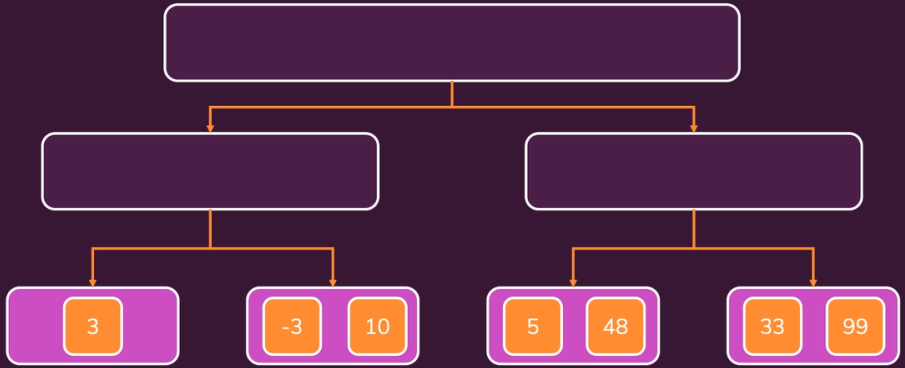
ACADEMIND

Merge Sort



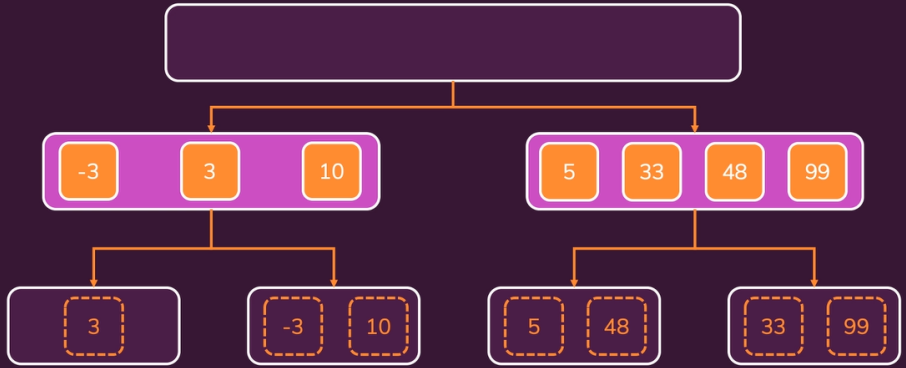
ACADEMIND

Merge Sort



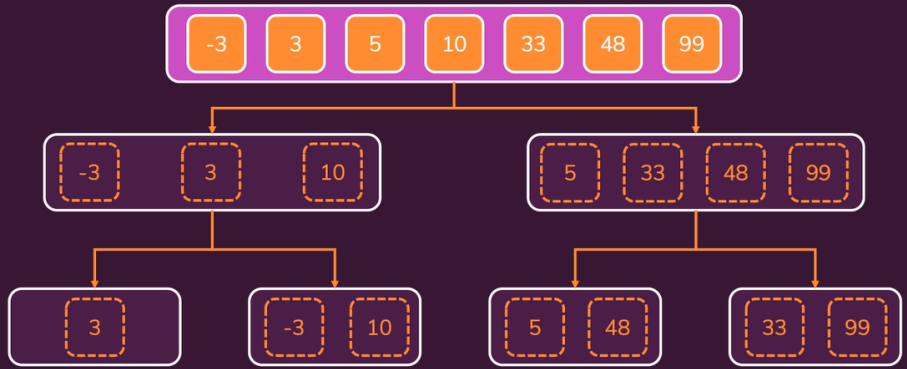
ACADEMIND

Merge Sort



ACADEMIND

Merge Sort



ACADEMIND

Merge Sort – Time Complexity

Best Case	Average Case	Worst Case
Items are sorted randomly	Items are sorted randomly	Items are sorted randomly
O(n * log n)	O(n * log n)	O(n * log n)

ACADEMIND

Space Complexity

Not just Runtime Performance

ACADEMIND

Module Content

What is Space Complexity?

Deriving Space Complexity

Examples

ACADEMIND

What is “Space Complexity”?

How much **space in memory** does your algorithm occupy?

All values in JavaScript are stored in memory

Especially arrays and objects can take up a bit more space

Generally, in JavaScript, **you won’t need to worry** about space complexity and memory too much though

ACADEMIND

Deriving Space Complexity

Find places where data (values) is stored "**permanently**" in your algorithm

Count how often such "permanently" stored values are being created (and kept around)

Determine how the number of values **depends on your "n"**

➔ O(n), O(1) etc. exists for space complexity as well

ACADEMIND

Examples

Algorithm	Space Complexity	Reason
Factorial (Loop)	O(1)	We operate on **one and the same** number, **no** new (“permanent”) value is created per iteration
Factorial (Recursive)	O(n)	A **new value is created for every nested function call** (the parameter received)
Linear Search	O(1)	**No** new ”permanent” values are created during the iteration
Binary Search	O(1)	**No** new ”permanent” values are created during the iteration

ACADEMIND

More Examples

Algorithm	Space Complexity	Reason
Bubble Sort	O(1)	**No** new "permanent" values are created during the iteration
Quicksort	O(n) *(O(log n) is possible)*	**Nested function calls** with new values being created
Merge Sort	O(n)	**Nested function calls** with new values being created

ACADEMIND

Sets (Array) Algorithms

Getting Tricky

ACADEMIND

Module Content

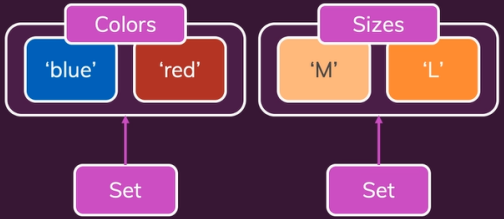
What are “Set (Array) Algorithms” All About?

Examples!

What are Set (Array) Algorithms About?

A Set is a collection of values (objects, numbers) which forms an entity itself

Example: Shirt manufacturing



Cartesian Product

Example: Shirt manufacturing

New Set: All variants that should be manufactured

ACADEMIND

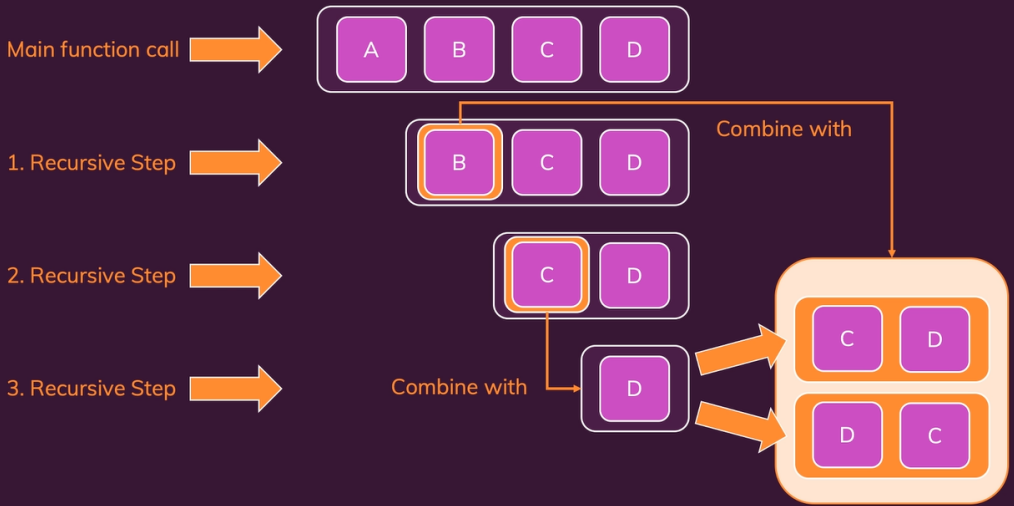
Cartesian Product – More Than 2 Sets

Permutations (With & Without Repetition)

An ordered combination of values.

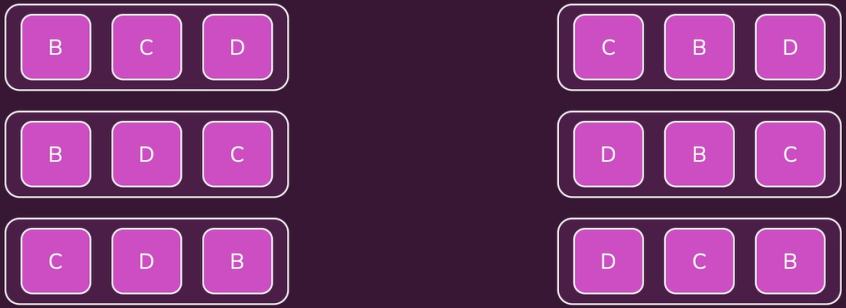
Without Repetition	With Repetition
Example: Todo list items	Example: Safe combination you want to set
['Walk dog', 'Clean toilet', 'Order food']	[1, 5, 9]
['Walk dog', 'Clean toilet', 'Order food'] ['Clean toilet', 'Walk dog', 'Order food'] ['Walk dog', 'Order food', 'Clean toilet'] ...	[1, 1, 9] [1, 1, 1] [9, 5, 9] ...

Recursive Permutations



ACADEMIND

Recursive Permutations



ACADEMIND

A Structured Problem Solving Approach

Solving ANY Problem

ACADEMIND

Module Content

How to approach problems

Examples!

ACADEMIND

Complex Algorithms Are Complex...

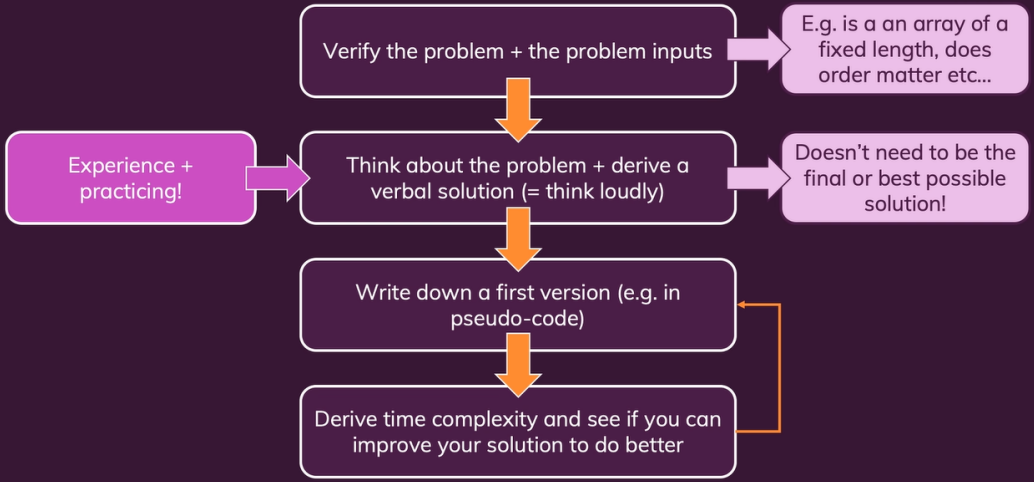
Coming up with the best possible algorithm can be very hard for tricky problems

That's why algorithms are popular in interviews: People want to see if you can solve problems

Good news: It's NOT the best possible solution that counts, it's your ability to come up with solutions

ACADEMIND

Solving Problems / Coming Up With Algorithms



Ways of Simplifying a Problem

Split into smaller sub-problems	Split arrays into chunks possibly combined with ... Recursion
Use console.log() or breakpoints to verify what's in your (temporary) variables	This is absolutely fine, also in interviews!
Use helper variables (e.g. helper arrays to store intermediate results)	Don't worry about space complexity, don't aim for the shortest "no overhead" solution right from the start

ACADEMIND

Practice Makes Perfect!

Finding good approaches to solve a problem **takes practice** – there is **no simple "blueprint"** that you can apply to every problem

Practice by diving into common algorithms and interview questions

ACADEMIND

The Knapsack Problem

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: 3, w: 3},
  {id: 'b', val: 6, w: 8},
  {id: 'c', val: 10, w: 3}
]

maxWeight = 8

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

ACADEMIND

Solving the Knapsack Problem

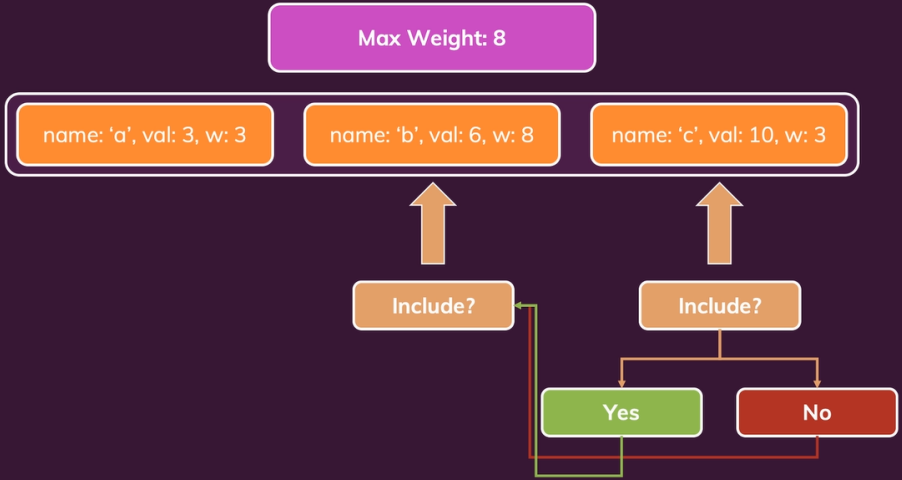
Verify inputs: Can items be used multiple times?

Derive a first (verbal) solution: We could derive all possible combinations and find the one with highest value and fitting weight

Write down a first version!

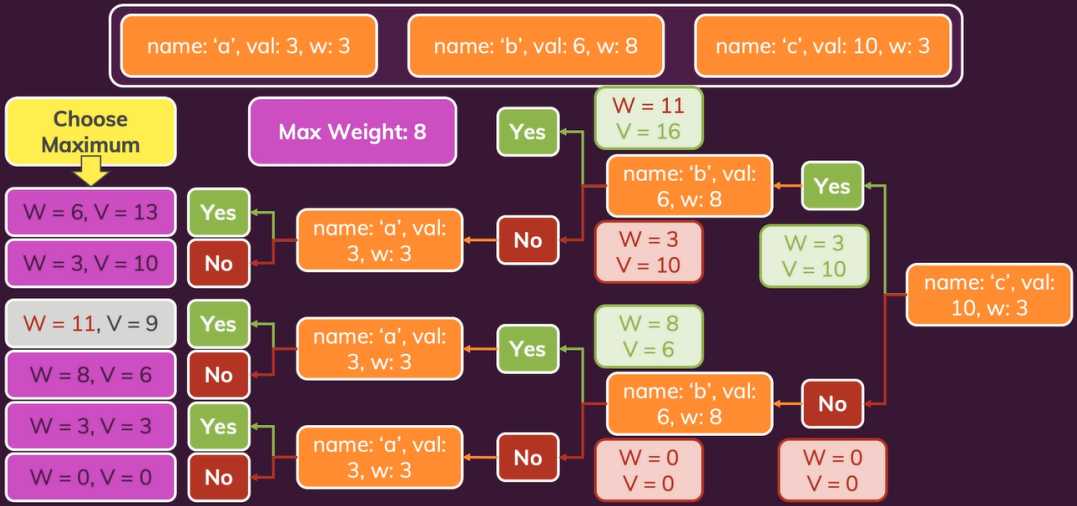
ACADEMIND

Let's Rethink!



ACADEMIND

We Evaluate All Possible Cases / Combinations



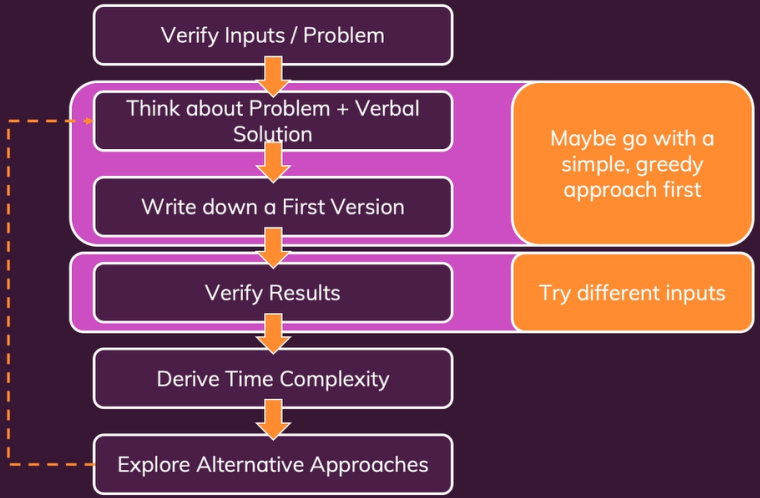
ACADEMIND

Greedy vs Dynamic Algorithms / Solutions

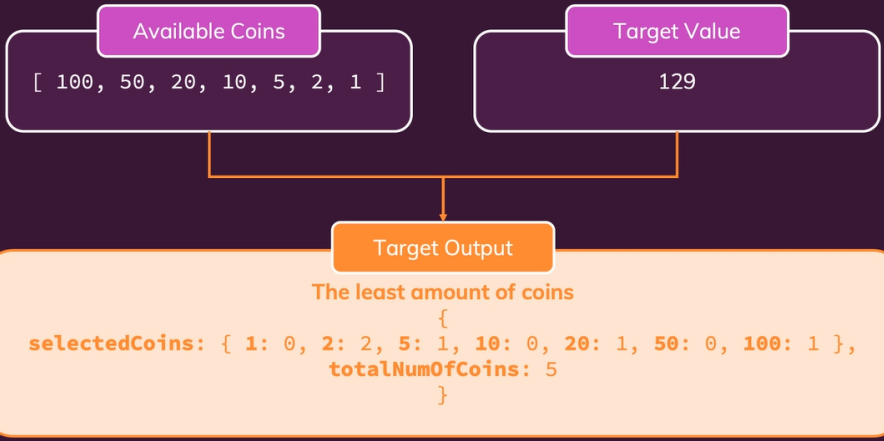
Greedy	Dynamic
Make best possible decision in every step and hope that it leads to the overall best solution	Evaluate all possible solutions and find overall best solution via comparison
Greedy algorithms often are faster to set up and come up with but they don't necessarily provide the best runtime and/ or result	"Divide and conquer" approach: Divide the problem into smaller, easy-to-solve subproblems

ACADEMIND

Our Final "Problem-Solving" Plan

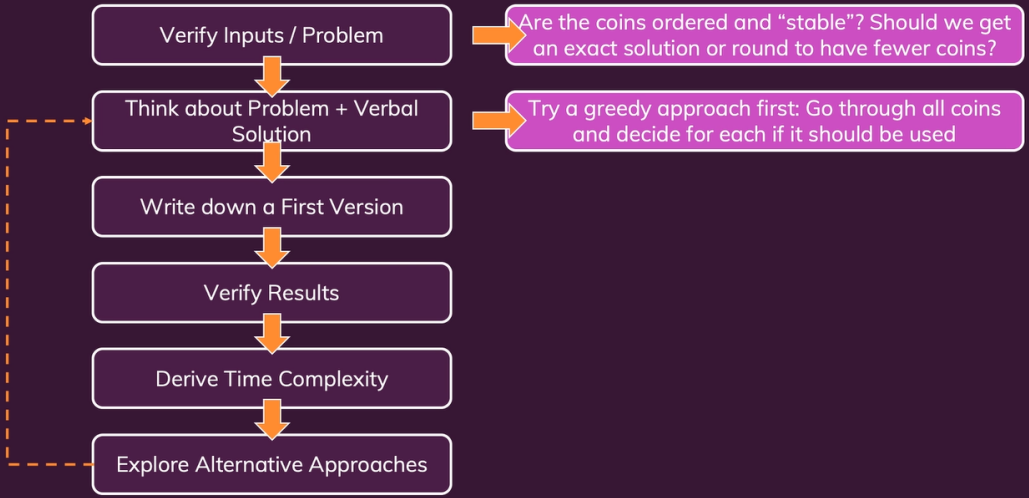


The Change Making Problem

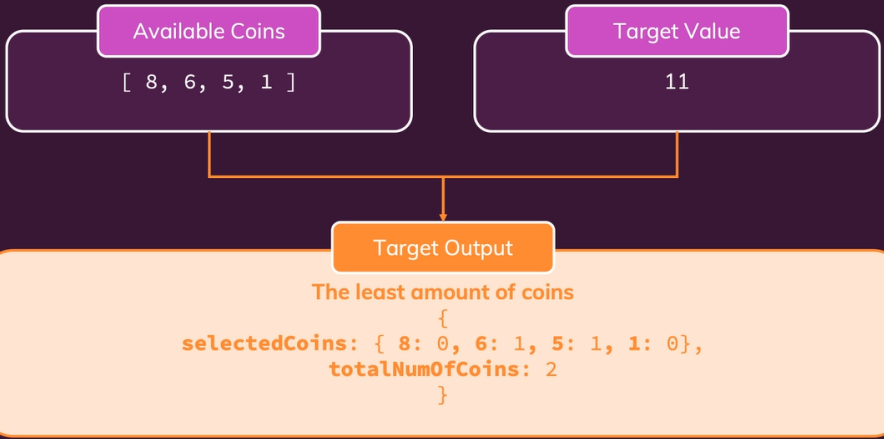


ACADEMIND

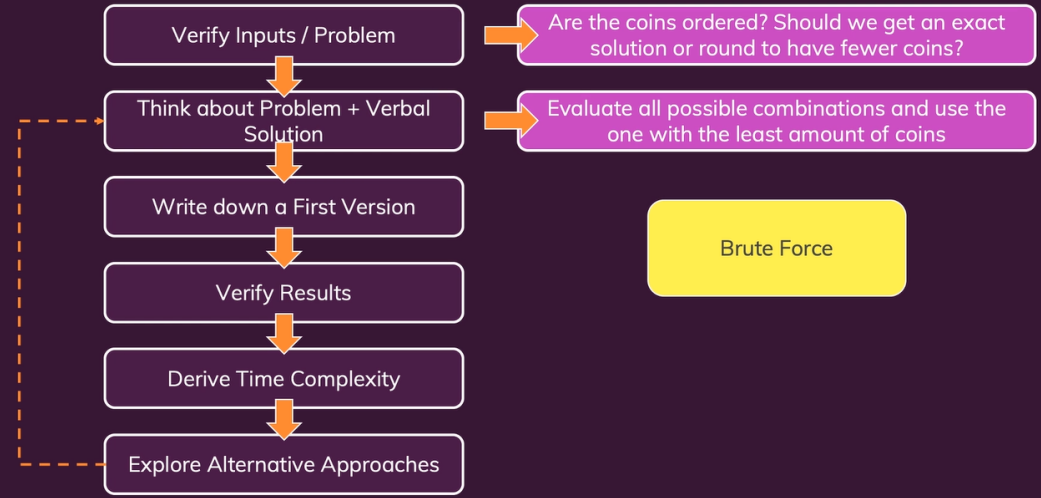
Change Making Problem: Our Plan



The More Difficult Change Making Problem



Change Making Problem: Adjusted Plan



ACADEMIND

How To Continue

Next Steps

ACADEMIND

Practice Algorithms!

You only get better by solving a lot problems and understanding a lot of algorithms

Good thing is: You find plenty of examples and explanations online – and with this course, you'll be able to understand them

Don't try to learn algorithms by heart! Understand common patterns, approaches etc. instead

ACADEMIND

Next Step: Data Structures

- “Data structures” are things like arrays and objects but typically, custom-built data structures are meant
- Custom data structures are built by combining default data structures like arrays and objects and enriching them with functionalities
- Such custom data structures can make solving some problems easier and/or more efficient
- If you understand algorithms, you’ll have an easy time understanding data structures