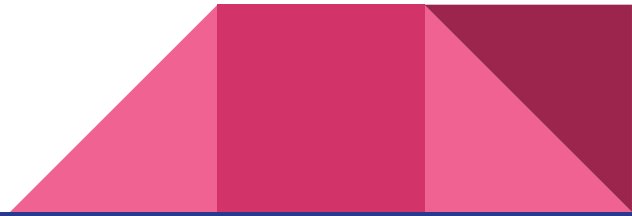
# Recursive Backtracking

# The Backtracking Checklist

- **Find what choice(s) we have at each step.** What different options are there for the next step?
- For each valid choice:
  - **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).
  - **Undo if after exploring.** Restore everything to the way it was before making this choice.
- Find the base case(s). What should we do when we are out of decisions?

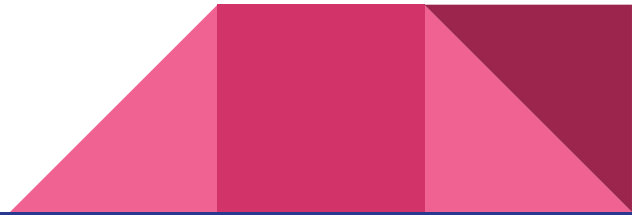


# groupSum problem

Given an array of ints, is it possible to choose a group of some of the ints, such that the group sums to the given target? This is a classic backtracking recursion problem. Once you understand the recursive backtracking strategy in this problem, you can use the same pattern for many problems to search a space of choices. Rather than looking at the whole array, our convention is to consider the part of the array starting at index **start** and continuing to the end of the array. The caller can specify the whole array simply by passing start as 0. No loops are needed -- the recursive calls progress down the array.

**public boolean groupSum(int start, int[] nums, int target) {}**

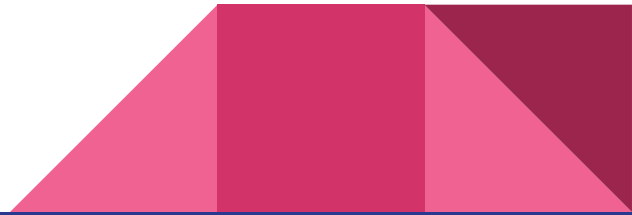
groupSum(0, [2, 4, 8], 10) → true
groupSum(0, [2, 4, 8], 14) → true
groupSum(0, [2, 4, 8], 9) → false

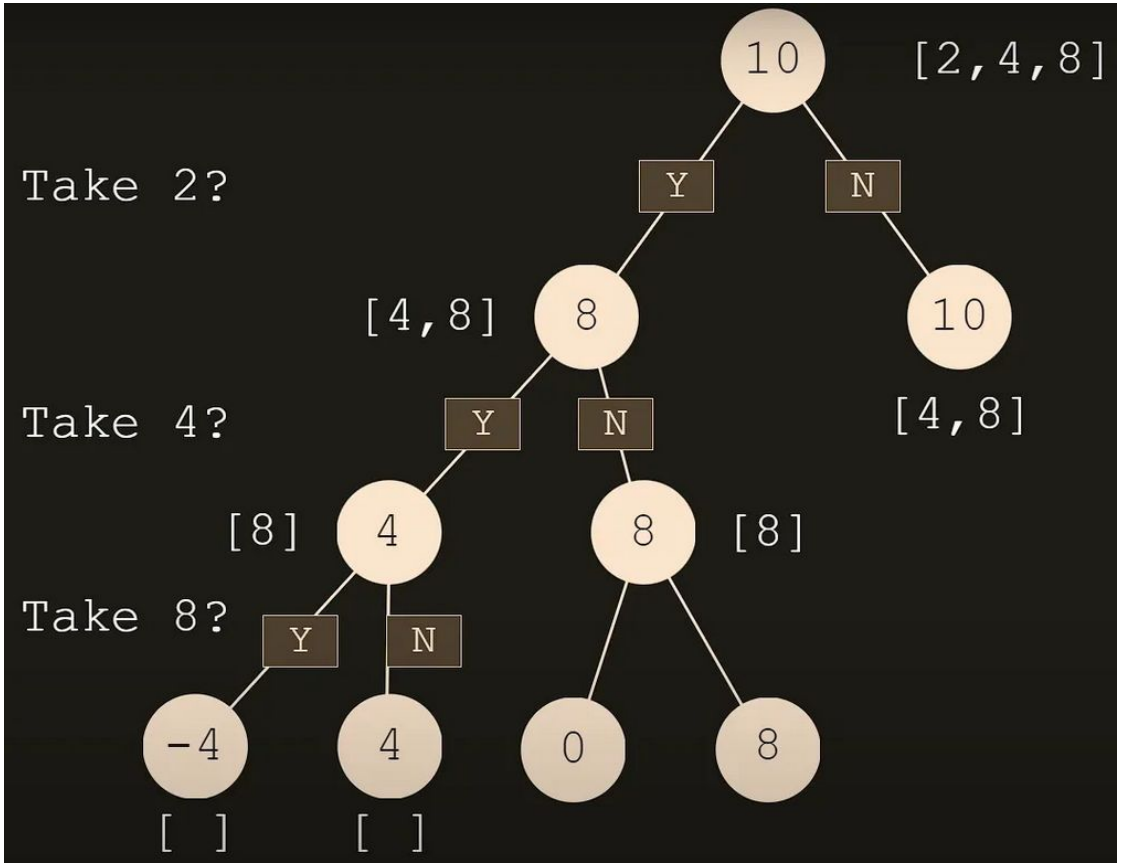


# groupSum - Hint

**public static boolean groupSum(int start, int[] nums, int target) {}**
groupSum(0, [2, 4, 8], 10) → true
groupSum(0, [2, 4, 8], 14) → true
groupSum(0, [2, 4, 8], 9) → false

The **base case** is when **`start >= nums.length`**. In that case, **return** true if **`target==0`**.
**Otherwise**, consider the element at **`nums[start]`**. The key idea is that there are only **2 possibilities** -- **`nums[start]` is chosen or it is not.**
**Make one recursive** call to see if a solution is possible **if nums[start] is chosen (subtract nums[start] from target in that call)**.
**Make another recursive** call to see if a solution is possible **if nums[start] is not chosen**.
**Return true** if either of the two recursive calls returns true.






groupSum(0, [2, 4, 8], 10) → true

Source: Daniel Sutantyo

# Coding Time!!!

**Save your work here:**

.../APCSA2/apcsa-assignments-spring-YourUsername/classwork/02_04_backtracking/Backtracking.java

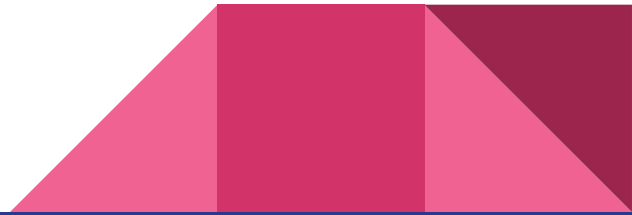
**Method signature:**

```
public static boolean groupSum(int start, int[] nums, int target) {

}
```

**Test Cases:**

```
groupSum(0, [2, 4, 8], 10) → true
groupSum(0, [2, 4, 8], 14) → true
groupSum(0, [2, 4, 8], 9)  → false
```

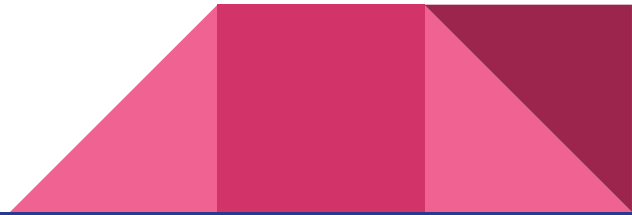


# CodingBat :o)

https://codingbat.com/home/jnovillo@stuy.edu/apcsa_recursion_backtracking_v1

Log in to your CodingBat account.

- Go to your prefs page
- Teacher Share section: type my email jnovillo@stuy.edu
- Memo section: type YourPeriod_YourLastName_YourFirstName, like this 01_smith_peter



# splitArray

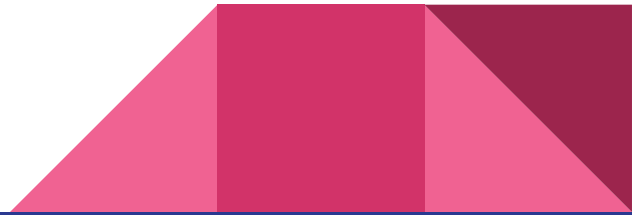
Given an array of ints, is it possible to divide the ints into two groups, so that the sums of the two groups are the same. Every int must be in one group or the other. Write a recursive helper method that takes whatever arguments you like, and make the initial call to your recursive helper from splitArray(). (No loops needed.)

splitArray([2, 2]) → true

splitArray([2, 3]) → false

splitArray([5, 2, 3]) → true

public boolean splitArray(int[] nums) { }

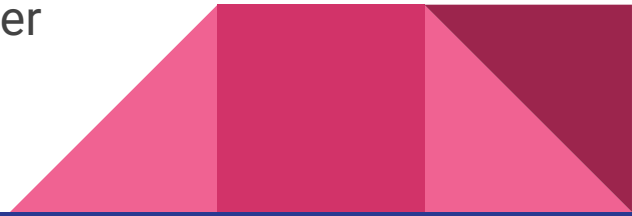


# Wrapper Method

A **wrapper method** is a method that serves as a simple entry point that **delegates** the real work to another method (usually a helper), often adding initial parameters that the caller shouldn't need to know about.

**Key characteristics:**

- Has a clean, simple public signature
- Does little to no logic itself
- Sets up initial values and calls the helper
- Hides the complexity/extra parameters from the caller



**splitArray([5, 2, 3])** **helper(int[] nums, int index, int group1, int group2)**