# Introdução

Saber um pouco de recursion, analise complexa, big O notation.

# Fibonacci memoization

Write a function ‘fib(n)’ that takes in a number as an argument. The function should return the n-th number of the Fibonacci sequence.

The 1st and 2nd number of the sequence is 1. To generate the next number of the sequence, we sum the previous two.

e.g.

n: 1, 2, 3, 4, 5, 6, 7, 8 , 9, 10

fib(n): 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

time complexity:

O(n) time

space complexity:

O(n) space

# Grid traveler

Say that you are a traveler on a 2D grid. You begin in the top-left corner and your goal is to travel to the bottom-right corner. You may only move down or right.

In how many ways can you travel to the foal on a grid with dimension m \* n?

Write a function ‘grid\_traveler’ that calculate this.

grid\_traveler(2, 3) = 3.

# Alvin’s memoization recipe

1. Make it work
   1. visualize the problem as a tree
   2. implement the tree using recursion
   3. test it, search for correctness first
2. Make it efficient.
   1. add a memo object
   2. add a base case to return memo values
   3. store return values into the memo

# Can sum

Write a function ‘can\_sum(target\_sum, numbers)’ that takes in a target\_sum and an array of numbers as arguments.

the function should return a boolean indicating whether or not it is possible to generate the target\_sum using numbers from the array.

You may use an element of the array as many times as needed.

You may assume that all input numbers are non-negative.

# How sum

Write a function ‘how\_sum(target\_sum, numbers)’ that takes in a target\_sum and an array of numbers as arguments.

The function should return an array containing any combination of elements that add up to exactly the target\_sum. If there is no combination that adds up to the target\_sum, then return null.

If there are multiple combinations possible, you may return any single one.

# Best sum

Write a function ‘best\_sum(target\_sum, numbers)’, that takes in target\_sum and an array of numbers as arguments.

The function should return an array containing the shortest combination of numbers that add up to exactly the target\_sum.

If there is a tie for the shortest combination, you may return any one of the shortest.

m = target sum

n = array length

brute force:

O(n^m \* m) time

O(m^2) space

memoized:

O(n \* m^2) time

O(m^2) space

can sum -> “Can you do it? yes/no” -> Decision Problem

how sum -> “How will you do it?” -> Combinatoric Problem

best sum -> “What is the ‘best’ way to do it?” -> Optimization Problem

# Can construct memoization

Write a function ‘’can\_construct(target, word\_bank)” that accepts a target string and a array of strings.

The function should return a boolean indication whether or not the ‘target’ can be constructed by concatenating elements of the ‘word\_bank’ array.

You may reuse elements of ‘word\_bank’ as many times asa needed.

# Count construct memoization

Write a function ‘count\_construct(target, word\_bank)’ that accepts a target string and an array of strings.

The function should return the number of ways that the ‘target’ can be constructed by concatenating elements of the ‘word\_ban’ array.

You may reuse elements of ‘word\_bank’ as many times as needed.

# All construct memoization

Write a function ‘all\_construct(target, word\_bank)’ that accepts a target string and an array of strings.

The function should return a 2D array containing all of the ways that the ‘target’ can be constructed by concatenating elements of the ‘word\_bank’ array. Each element of the 2D array should represent one combination that constructs the ‘target’.

You may reuse elements of ‘word\_bank’ as many times as needed.

# Tabulation recipe

* visualize the problem as a table
* size the table based on the inputs
* initialize the table with default values
* seed the trivial answer into the table
* iterate through the table
* fill further positions based on the current position