Hoy:

- Algoritmos voraces
- Programación dinámica

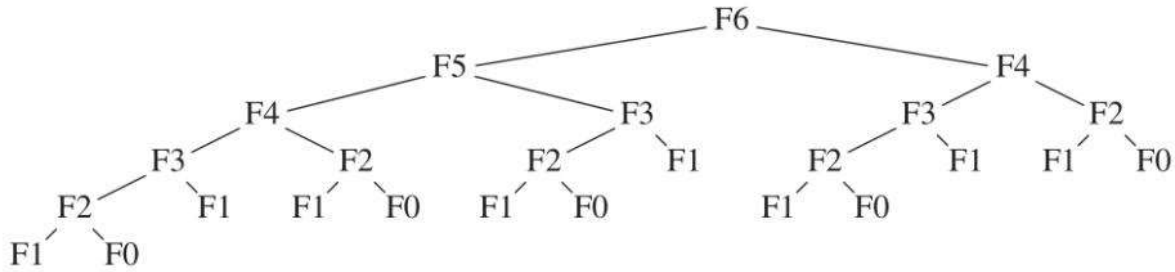
Programación dinámica:
Ejemplo Fibonacci (una de las soluciones)

Problemas matemáticos definidos recursivamente, se implementan recursivamente de forma muy sencilla....
.... pero son muy ineficaces

Ejemplo Fibonacci recursivo O(2^n)

Recursivo:

```
/**
 * Compute Fibonacci numbers as described in Chapter 1.
 */
long long fib( int n )
{
    if( n <= 1 )
        return 1;
    else
        return fib( n - 1 ) + fib( n - 2 );
}
```



Programación dinámica:

- Habitualmente optimización
- Rellena resultados intermedios en tabla
- Enfoque ascendente
- Principio de optimalidad
  - Cómo trabajar
    - Ecuación recurrente
    - Definir tablas y cómo rellenarlas

Principio de optimalidad
ABC camino mínimo
AB y BC son Caminos mínimos

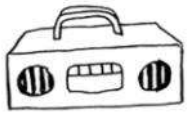
Enfoque de programación dinámica

```
long long fibonacci( int n )
{
    if( n <= 1 )
        return 1;

    long long last = 1;
    long long last nextToLast = 1;
    long long answer = 1;

    for( int i = 2; i <= n; ++i )
    {
        answer = last + nextToLast;
        nextToLast = last;
        last = answer;
    }
    return answer;
}
```

Problema de la mochila



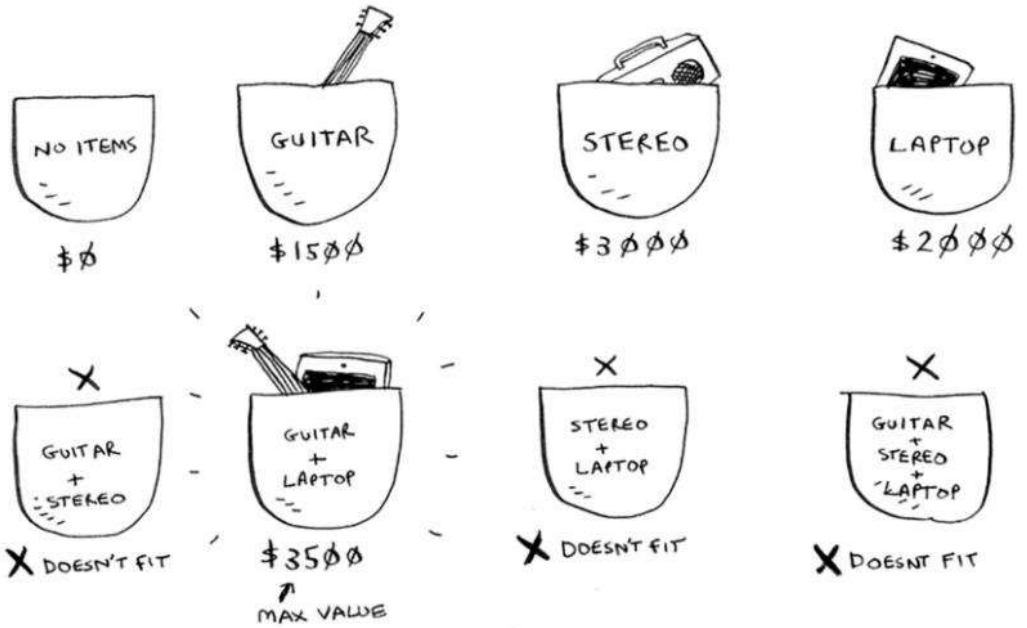
STEREO
$3000
4lbs

LAPTOP
$2000
3 lbs

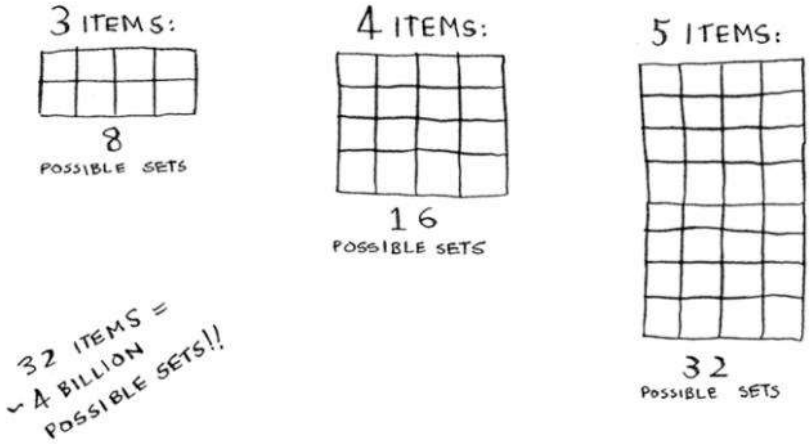
GUITAR
$1500
1 lbs

Puedes llevar 4Lb

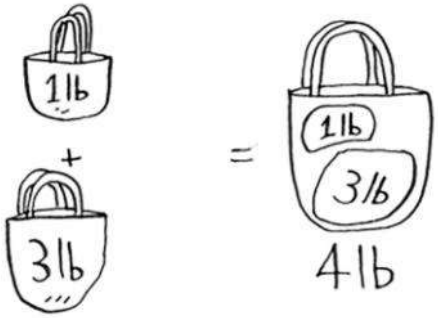
1er enfoque, probar todas las soluciones

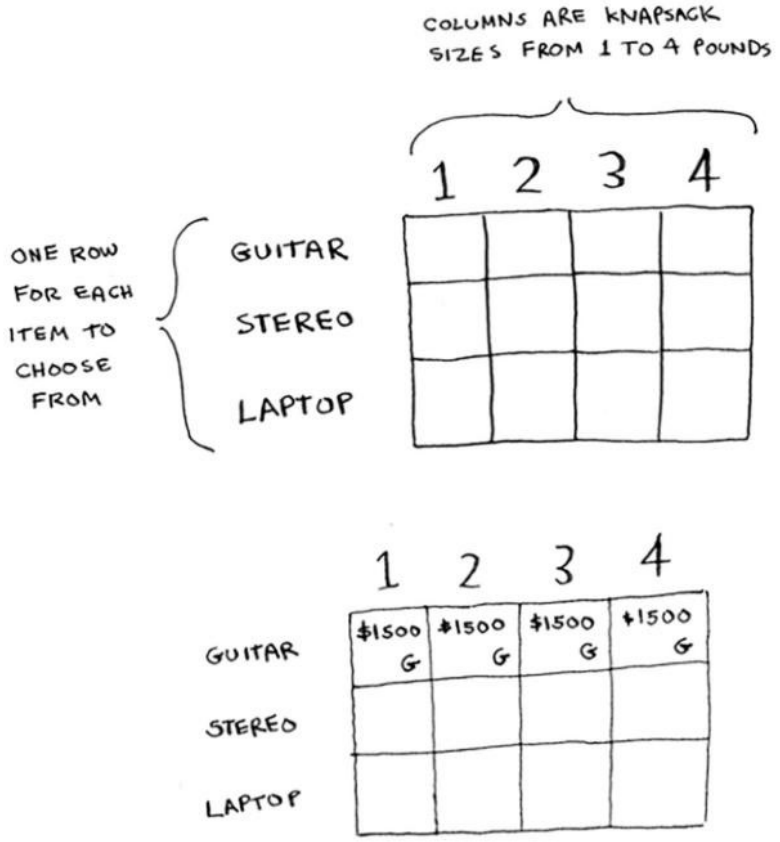


Muy costoso

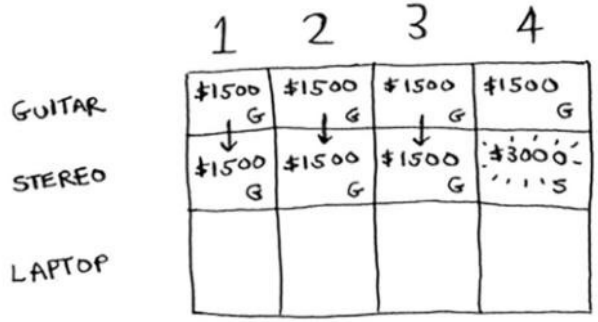

3 ITEMS:
8
POSSIBLE SETS
4 ITEMS:
16
POSSIBLE SETS
5 ITEMS:
32
POSSIBLE SETS
32 ITEMS =
~4 BILLION
POSSIBLE SETS!!

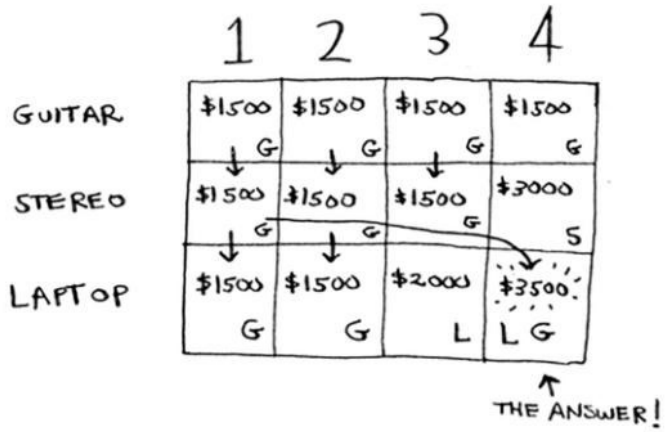
Programación dinámica, soluciona primero problemas más pequeños




COLUMNS ARE KNAPSACK SIZES FROM 1 TO 4 POUNDS
1 2 3 4
ONE ROW FOR EACH ITEM TO CHOOSE FROM
GUITAR
STEREO
LAPTOP

1 2 3 4
GUITAR
$1500 G
$1500 G
$1500 G
$1500 G
STEREO
LAPTOP


1
2
3
4
GUITAR
STEREO
LAPTOP
$1500
G
$1500
G
$1500
G
$1500
G
$1500
G
$1500
G
$1500
G
$300
5



ROW COLUMN

CELL[i][j] = max of { 1. THE PREVIOUS MAX (VALUE AT CELL [i-1][j])

VS

2. VALUE OF CURRENT ITEM + VALUE OF THE REMAINING SPACE

↑

CELL[i-1][j − ITEM'S WEIGHT] }

Items fraccionables

Voraces (Greedy):
Ejemplos Dijsktra, Prim, Kruskal

Habitualemte problemas de optimización
Trabajan con mínimos locales, con la esperanza de llegar a un mínimo global.
Sigue el principio: "take what you can get now"
Pueden producir soluciones subóptimas

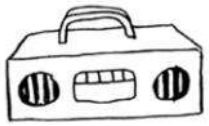
Mochila voraz

The knapsack can hold 35 pounds.

You're trying to maximize the value of the items you put in your knapsack. What algorithm do you use?

Again, the greedy strategy is pretty simple:

1. Pick the most expensive thing that will fit in your knapsack.
2. Pick the next most expensive thing that will fit in your knapsack. And so on.



STEREO
$3000
30lbs

LAPTOP
$2000
20lbs

GUITAR
$1500
15lbs

Items fraccionables

Resumen

- Dinámica
  - Habitualmente optimización
  - Rellena resultados intermedios en tabla
  - Enfoque ascendente
  - Principio de optimalidad
  - Cómo trabajar
    - Ecuación recurrente
    - Definir tablas y cómo rellenarlas
- Algoritmos voraces (greedy)
  - Habitualmente optimización
  - Cómo trabajar
    - Se parte de solución vacía
    - De entre los candidatos se añade uno
    - Continuar hasta solución

Problema del cambio de monedas:

Disponemos de monedas con valores de 1, 2, 5, 25, 50 y 100 céntimos de euro (c€). Construir un algoritmo que **dada una cantidad P devuelva el cambio** con monedas de estos valores, **usando un conjunto mínimo de monedas.**

P. ej.: para devolver 2.89 €: 2 monedas de 1 €, 1 moneda de 50 c€,
1 moneda de 25 c€, 2 monedas de 5 c€ y 2 monedas de 2 c€.

```
// Return minimum number of coins to make change.
// Simple recursive algorithm that is very inefficient.
int makeChange( const vector<int> & coins, int change )
{
    int minCoins = change;

      // Look for exact match with any single coin.
    for( int i = 0; i < coins.size( ); i++ )
        if( coins[ i ] == change )
            return 1;

      // No match; solve recursively.
    for( int j = 1; j <= change / 2; j++ )
    {
        int thisCoins = makeChange( coins, j )
                      + makeChange( coins, change - j );
        if( thisCoins < minCoins )
            minCoins = thisCoins;
    }

    return minCoins;
}
```

```
// Dynamic programming algorithm for change-making problem.
// As a result, the coinsUsed array is filled with the minimum
// number of coins needed for change from 0->maxChange and
// lastCoin contains one of the coins needed to make the change.
void makeChange( const vector<int> & coins, int maxChange,
                 vector<int> & coinsUsed, vector<int> & lastCoin )
{
    int differentCoins = coins.size( );
    coinsUsed.resize( maxChange + 1 );
    lastCoin.resize( maxChange + 1 );

    coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;
    for( int cents = 1; cents <= maxChange; cents++ )
    {
        int minCoins = cents, newCoin = 1;
        for( int j = 0; j < differentCoins; j++ )
        {
            if( coins[ j ] > cents )   // Can't use coin j
                continue;
            if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins )
            {
                minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
                newCoin  = coins[ j ];
            }
        }

        coinsUsed[ cents ] = minCoins;
        lastCoin[ cents ]  = newCoin;
    }
}
```

Cambio de monedas: ¿Versión voraz?

¿Tiene solución?
¿Es optima?

Ejercicio:

```
bool CambioMondedas( const vector<moneda> & monedas, double cantidad,
                                                     vector<moneda> & cambio)
```