Advanced Programming
INFO135

Lecture 10: Dynamic Programming

Mehdi Elahi

University of Bergen (UiB)

timeit

- **`timeit:`** a built-in Python library used to measure the execution time

```
import timeit


def my_function():
    pass


print(timeit.timeit(my_function, number=10000))
```

[Output:]

```
0.0008361259999999995
```

timeit: another example

timeit

```
import timeit

my_code = '''
def my_function(n):
    print(n)

my_function(9)
'''

print(timeit.timeit(my_code, number=10000))
```

the code to run and to assess

[Output:]

.

.

.

0.04555064500000001

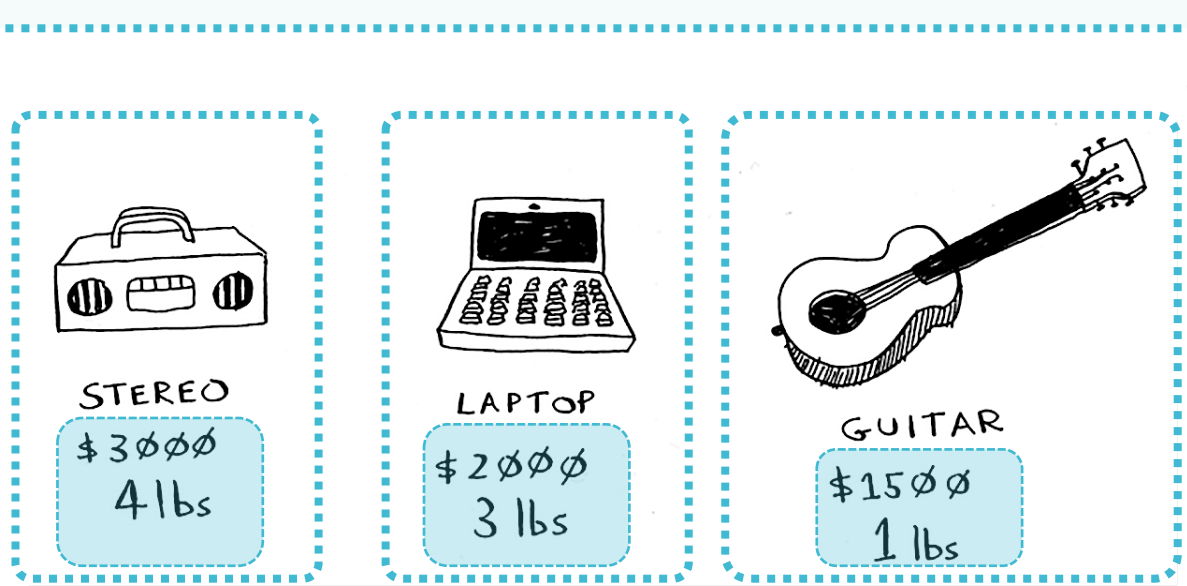
timeit

`timeit.timeit(stmt,setup,timer,n)`

- **`stmt`** is the statement you want to measure.
- **`setup`** is the code that you run before running the `stmt`.
- **`timer`** is the `timeit` object.
- **`n`** is the number of executions you'd like to run the `stmt`.

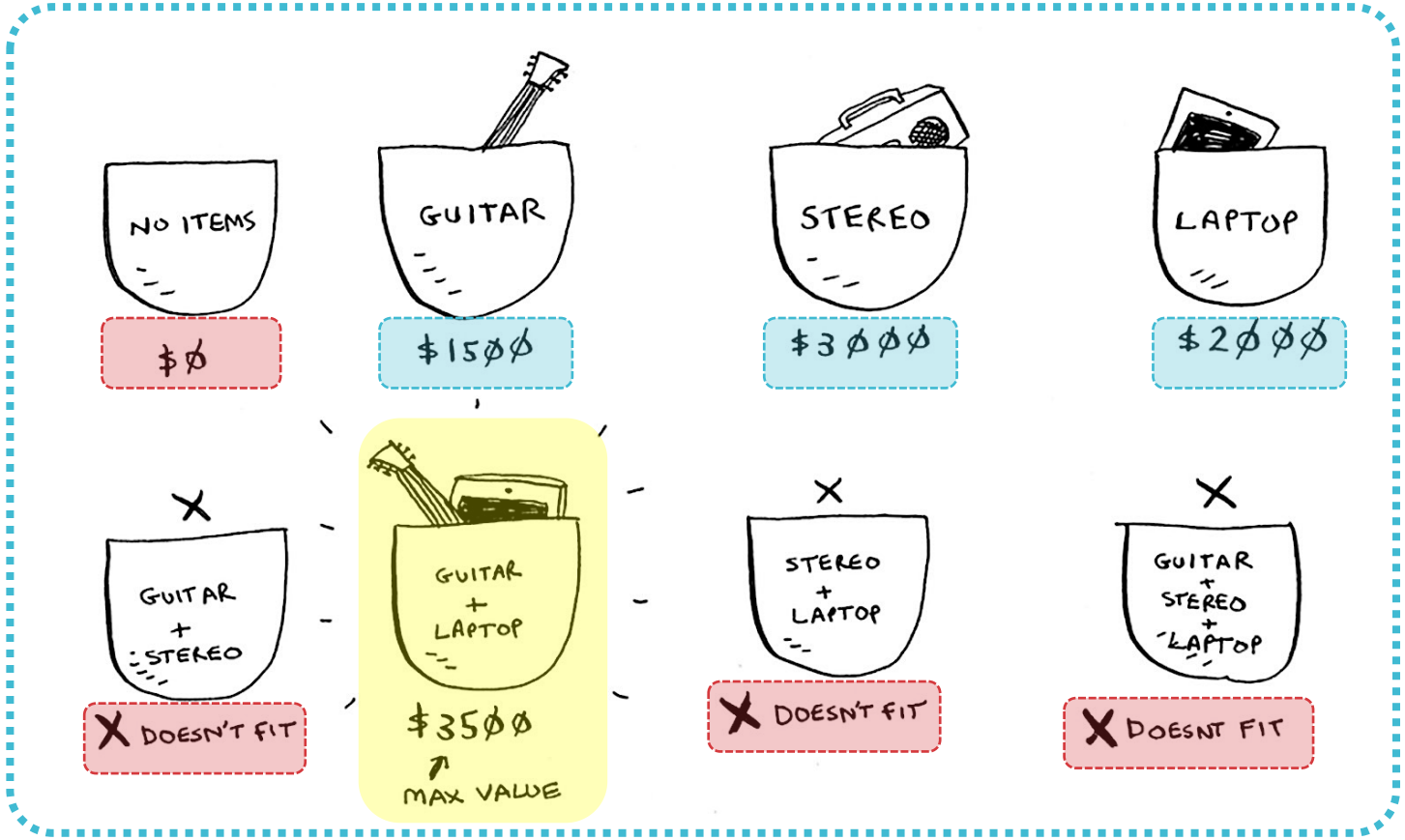
Dynamic Programming

- Suppose a thief is in a store **to steal as many items as possible**.
- It can only take what fits in ***knapsack*** with fixed capacity (lb). Note: **1 lb** ≈ 0.45 Kg
- What algorithm can **maximize the value ($)** of the items that can all be put in the knapsack?



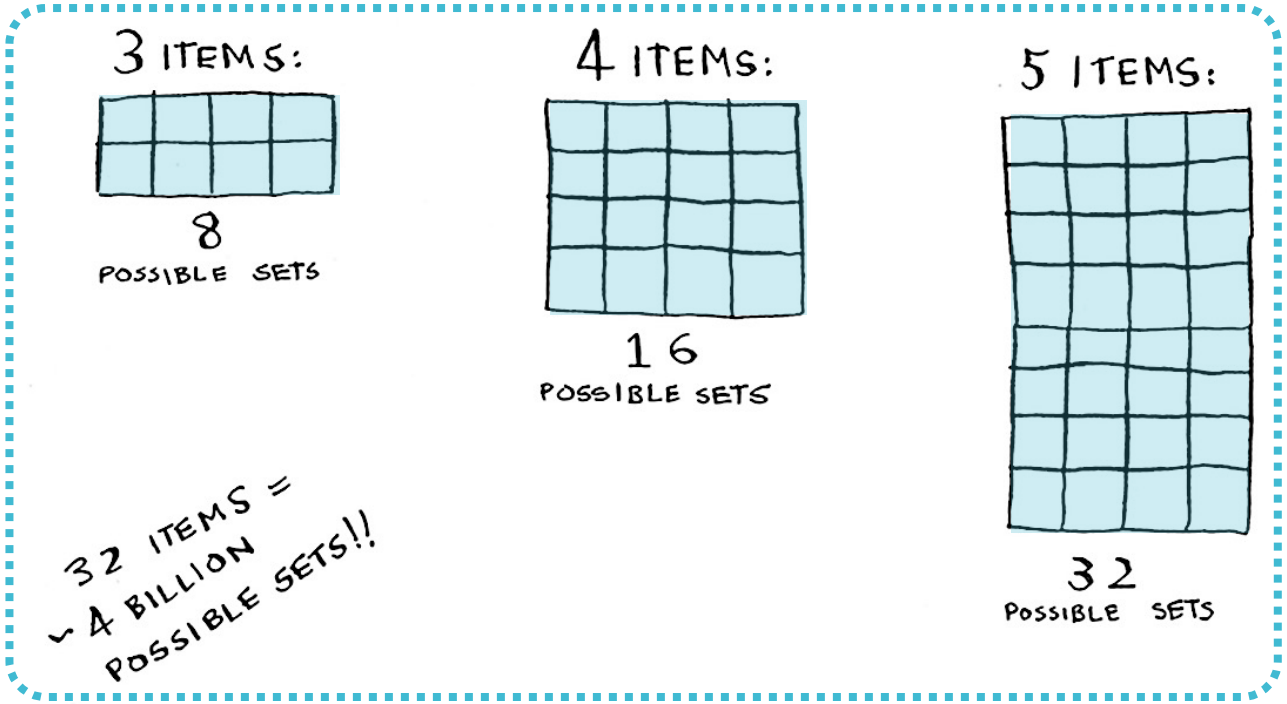
Dynamic Programming

- The **algorithm to solve** the problem will:
  1. compute **every possible** set of items.
  2. find the set that gives the **maximum value.**



Dynamic Programming

- This works, but it's **really slow**.
  - for 3 items --> 8 possible sets.
  - for 4 items --> 16 sets.
  - for 32 items --> 4 billion sets!
- This algorithm takes **$O(2^n)$ time**.



Dynamic Programming

- In the previous lecture, we learned about **greedy algorithms**.

- So we can also use such an algorithm in this scenario to give us an **approximation** of the solution.

Dynamic Programming

- The **greedy strategy** can do that by:
  1) picking the **most expensive** item that fits in the knapsack.
  2) then, picking the next most expensive thing that fits in the knapsack and **keep doing**.

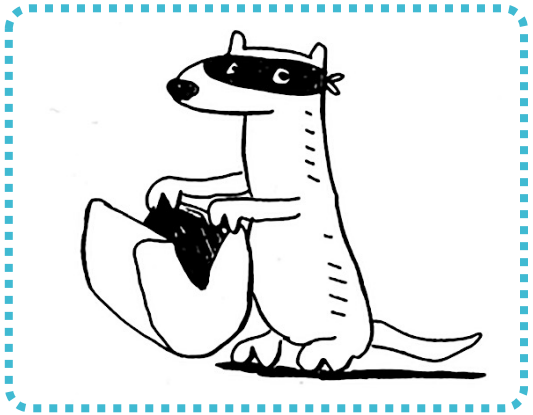
- But there is a problem!

Dynamic Programming

- Such an algorithm **doesn't always** give the **optimal** solution.
- But it gets you **pretty close**.
- Probably a thief does n't care for perfection.
- "Pretty good" is good enough!

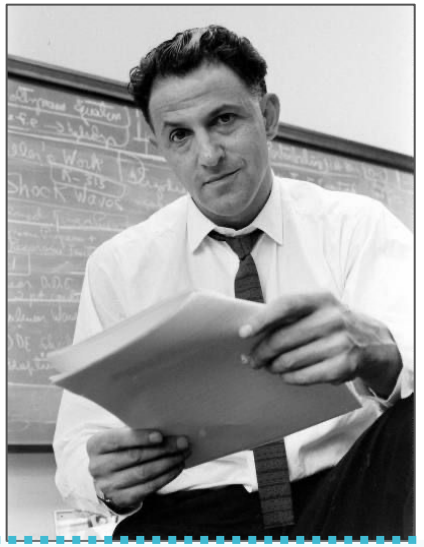
Dynamic Programming

- We can try another approach called **Dynamic Programming.**



Dynamic Programming

- **Developed by** Richard Bellman
- **in 1950**

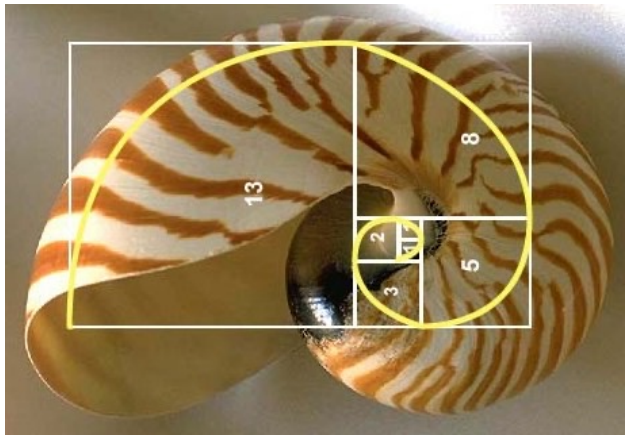


- **Dynamic programming:** the idea is to break down a problem into sub-problems, solving them and and storing theire sub-solutions.
- This way, each sub-problem is calculated only once.

- Approaches in Dynamic Programming:
  a) **Memoisation** (Top-Down)
  b) **Tabulation** (Bottom-Up)

Dynamic Programming

- Lets check a famous problem: **Fibonacci sequence**
- We briefly talked about it before.



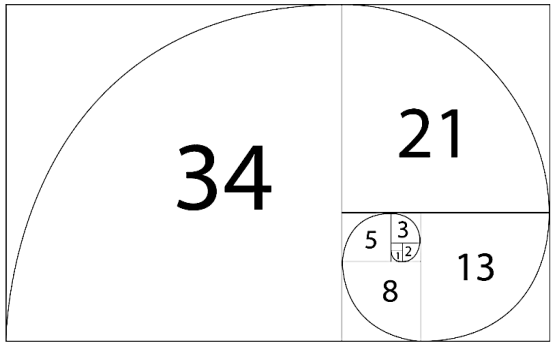
Dynamic Programming

- **Fibonacci sequence:** is a sequence of integer numbers in which each subsequent value is the *sum* of the two previous values.

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2), & \text{if } n > 1 \\ n, & \text{if } n = 1 \text{ or } n = 0 \end{cases}$$

- The first 11 numbers of the sequence are:

1, 1, 2, 3, 5, 8, 13, 21, 34, …



Dynamic Programming

❖ **Fibonacci sequence** with recursion:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)


print('The 40th Fibonacci: ', fib(40))
```

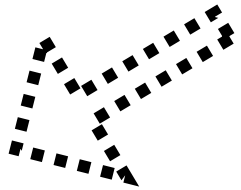
[output:]

```
The 40th Fibonacci:  102334155
```

Dynamic Programming

❖ **Fibonacci sequence** with recursion execution time:

```
import timeit

my_code = '''
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
print('The 40th Fibonacci : ', fib(40))
'''
time = timeit.timeit(stmt=my_code, number=1)
print('It takes ', time, ' sec.')
```



[output:]

```
The 40th Fibonacci :  102334155
It takes  40.705187714000004  sec.
```

Dynamic Programming

- In **Fibonacci sequence** problem, we:

1. **Divide** the problem into smaller sub-problems of the same type.
2. **Conquer**: solve sub-problems recursively.
3. **Combine**: Combine all the sub-problems to create a solution to the original problem.

- Dynamic programming has an extra step added to step 2, called **memoisation**.

Dynamic Programming

- **Memoization:** an optimization technique used primarily to speed up programs by
  - **storing the results** of expensive function calls
  - returning the stored result when the same inputs **occur again**.

- Memoisation is a **Top-Down** process of **storing sub-solutions**.

Dynamic Programming

❖ **Fibonacci sequence** with memoisation :

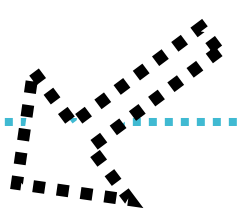
```
def fib2(n):
    memo = [0] * (n + 1)
    memo[0], memo[1] = 0, 1
    for i in range(2, n + 1):
        memo[i] = memo[i - 1] + memo[i - 2]
    return memo[n]


print('The 40th Fibonacci: ', fib2(40))
```

[output:]

```
The 40th Fibonacci:  102334155
```

Dynamic Programming

❖ **Fibonacci sequence** with memoisation :




```
def fib2(n):
    memo = [0] * (n + 1)
    memo[0], memo[1] = 0, 1
    for i in range(2, n + 1):
        memo[i] = memo[i - 1] + memo[i - 2]
    return memo[n]


print('The 40th Fibonacci: ', fib2(40))
```

[output:]

```
The 40th Fibonacci:  102334155
```

Dynamic Programming

❖ **Fibonacci sequence** with memoisation :

```
def fib2(n):
    memo = [0] * (n + 1)
    memo[0], memo[1] = 0, 1
    for i in range(2, n + 1):
        memo[i] = memo[i - 1] + memo[i - 2]
    return memo[n]
```

f(n) =
f(n-1) + f(n-2)

```
print('The 40th Fibonacci: ', fib2(40))
```

[output:]

```
The 40th Fibonacci:  102334155
```

Dynamic Programming

❖ **Fibonacci sequence** with memoisation:

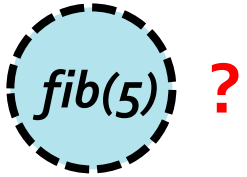
```
import timeit

my_code = '''
def fib2(n):
    memo = [0] * (n + 1)
    memo[0], memo[1] = 0, 1
    for i in range(2, n + 1):
        memo[i] = memo[i - 1] + memo[i - 2]
    return memo[n]
print('The 40th Fibonacci: ', fib2(40))
'''
time = timeit.timeit(stmt=my_code, number=1)
print('It takes ', time, ' sec.')
```

[output:]

```
The 40th Fibonacci:  102334155
It takes  4.261399999999846e-05  sec.
```

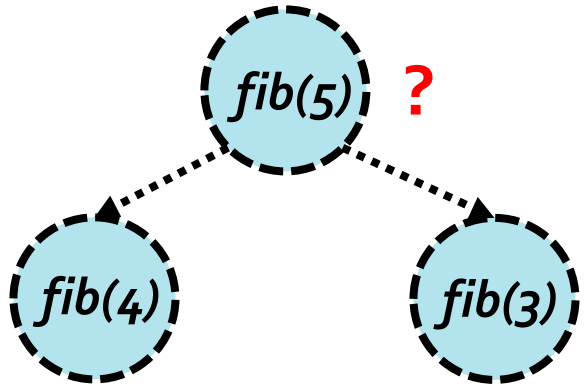
Dynamic Programming

- **Fibonacci sequence** recursion tree.
- Lets compute ***fib(5)***



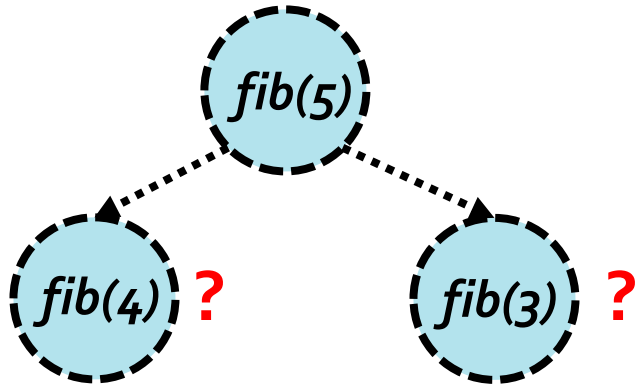
Dynamic Programming

- Lets compute ***fib(5)*** = ***fib(4)*** + ***fib(3)***



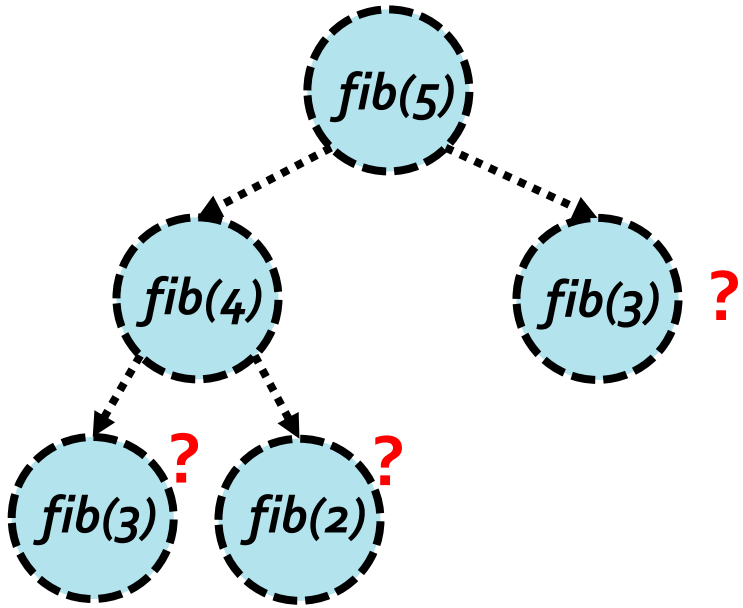
Dynamic Programming

- So we need to compute:
  - *fib(4)*
  - *fib(3)*



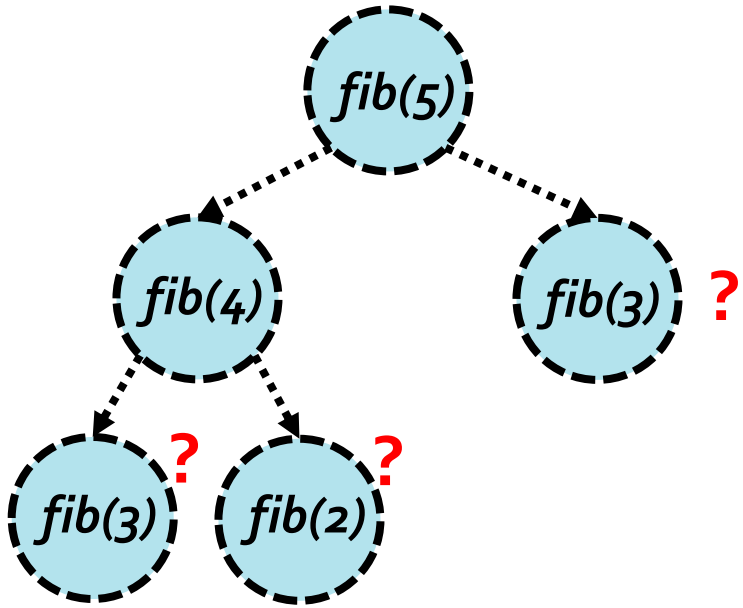
Dynamic Programming

- So we need to compute:
  - ***fib(4) = fib(3) + fib(2)***



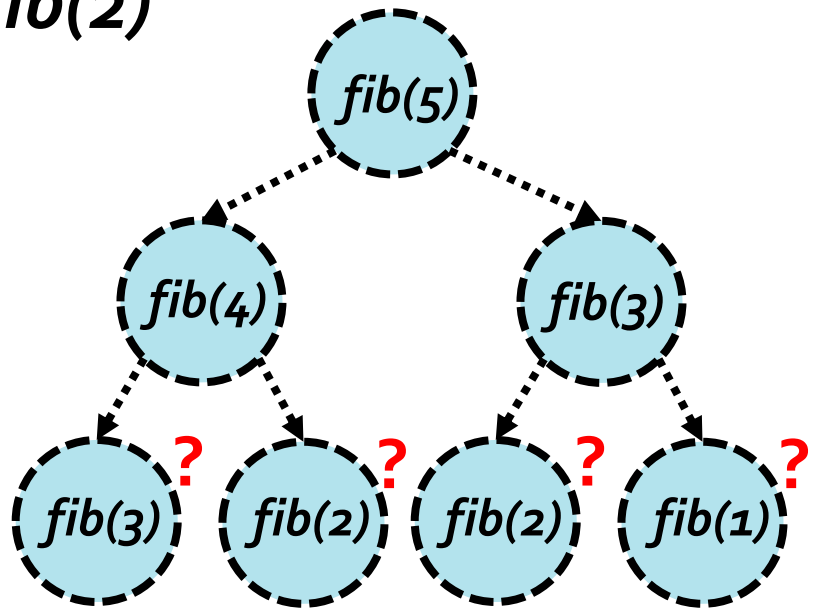
Dynamic Programming

- So far we need compute:
  - *fib(4)*
  - again *fib(3)*
  - *fib(2)*



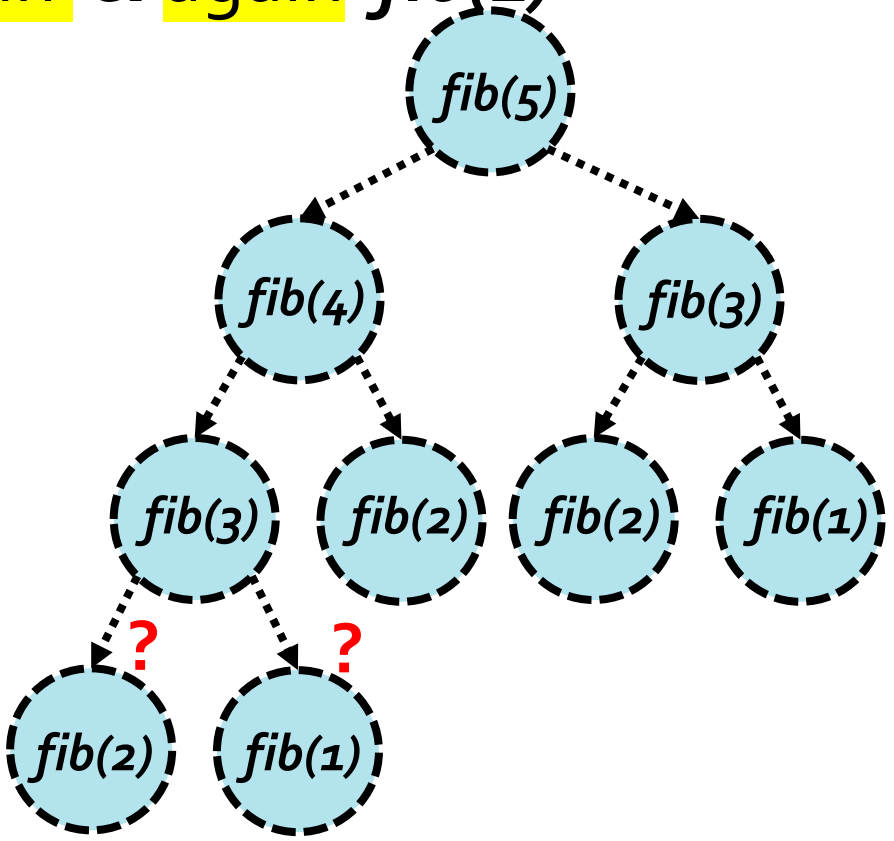
Dynamic Programming

- So far we need compute:
  - ***fib(4)***
  - again ***fib(3)***
  - again ***fib(2)***
  - ***fib(1)***



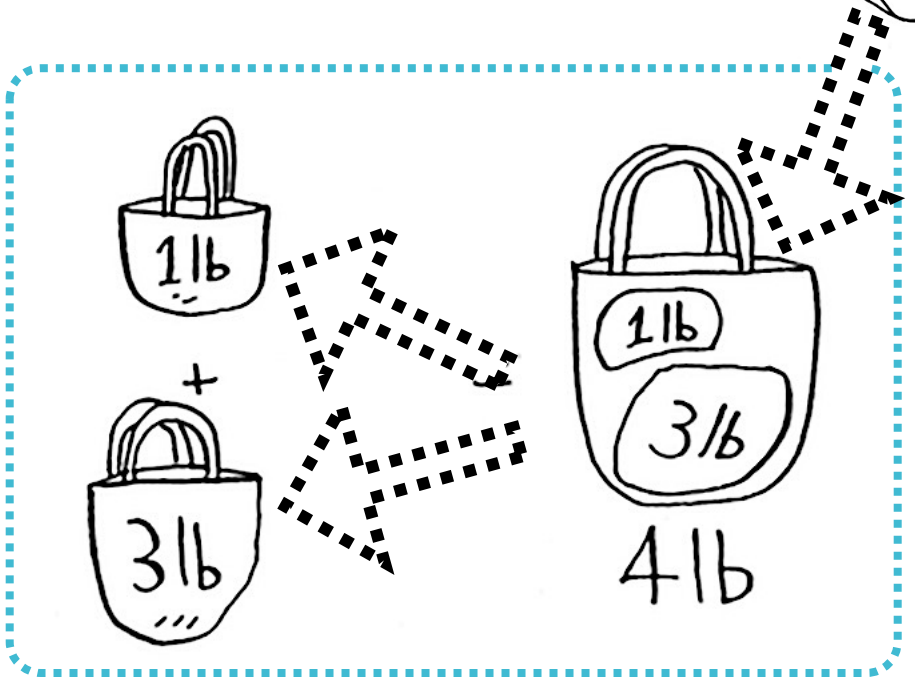
Dynamic Programming

- If we don't store, we need to compute:
  - …
  - again & again ***fib(2)***
  - again & again ***fib(1)***



Dynamic Programming

- Now lets check the **Knapsack problem**.
- A famous version is called 0-1 Knapsack Problem**.**
- We can solve it with **naïve Recursion**



Knapsack problem with **Recursion**

Dynamic Programming

```
def knapsack_rec(capacity, weights, values, n):
    if n == 0 or capacity == 0:
        return 0
```

base case:
no capacity
or no item

Knapsack problem with **Recursion**

Dynamic Programming

```
def knapsack_rec(capacity, weights, values, n):
    elif weights[n - 1] > capacity:
        return knapsack_rec(capacity, weights, values, n - 1)
```

item too heavy, we do not put it in knapsack

we check another item

Knapsack problem with **Recursion**

Dynamic Programming

```
def knapsack_rec(capacity, weights, values, n):
```

choice 1: if we don't choose item to put it in knapsack

```
    else:
        tmp1 = knapsack_rec(capacity, weights, values, n - 1)

        tmp2 = values[n - 1] + \
               knapsack_rec(capacity - weights[n - 1],
                            weights, values, n - 1)
```

choice 2: if we choose the item to put in the knapsack

Knapsack problem with **Recursion**

Dynamic Programming

```
def knapsack_rec(capacity, weights, values, n):

    else:
        tmp1 = knapsack_rec(capacity, weights, values, n - 1)

        tmp2 = values[n - 1] + \
               knapsack_rec(capacity - weights[n - 1],
                            weights, values, n - 1)

        return max(tmp1, tmp2)
```

we **compare two choices** and decide the one with the max value

Knapsack problem with **Recursion**

Dynamic Programming

```
def knapsack_rec(capacity, weights, values, n):
    if n == 0 or capacity == 0:
        return 0

    elif weights[n - 1] > capacity:
        return knapsack_rec(capacity, weights, values, n - 1)

    else:
        tmp1 = knapsack_rec(capacity, weights, values, n - 1)

        tmp2 = values[n - 1] + \
               knapsack_rec(capacity - weights[n - 1],
                            weights, values, n - 1)

        return max(tmp1, tmp2)
```

Dynamic Programming

Testing our implementation

```
item_val = [3000, 2000, 1500]

item_wt = [4, 3, 1]
total_cap = 4
n_items = len(item_val)

print('Max value to put in knapsack of capacity W:')
print(knapsack_rec(total_cap, item_wt, item_val, n_items), '$')
```

prices [$]

weights [lb]

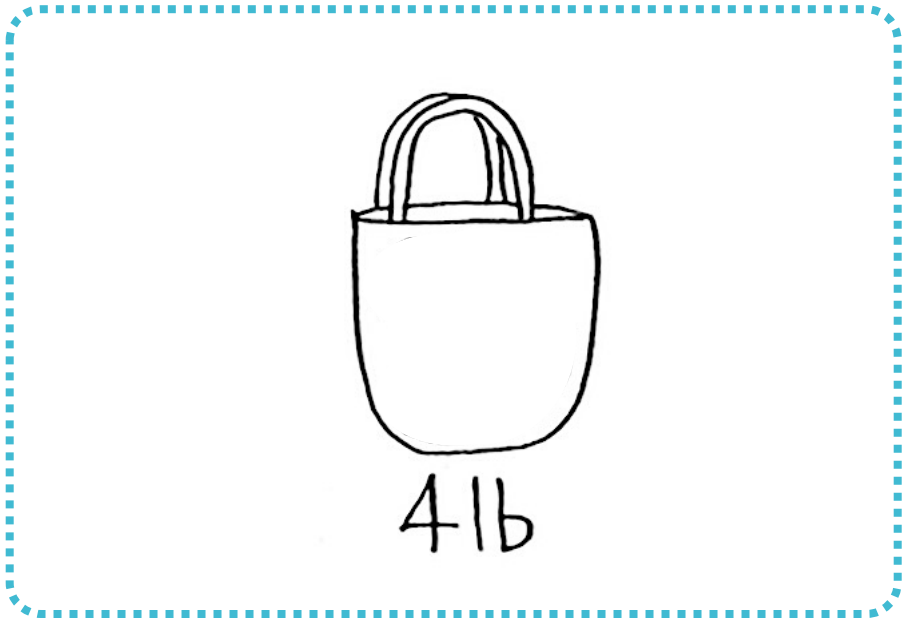
[Output:]

```
Max value to put in knapsack of capacity W:
3500 $
```

Dynamic Programming

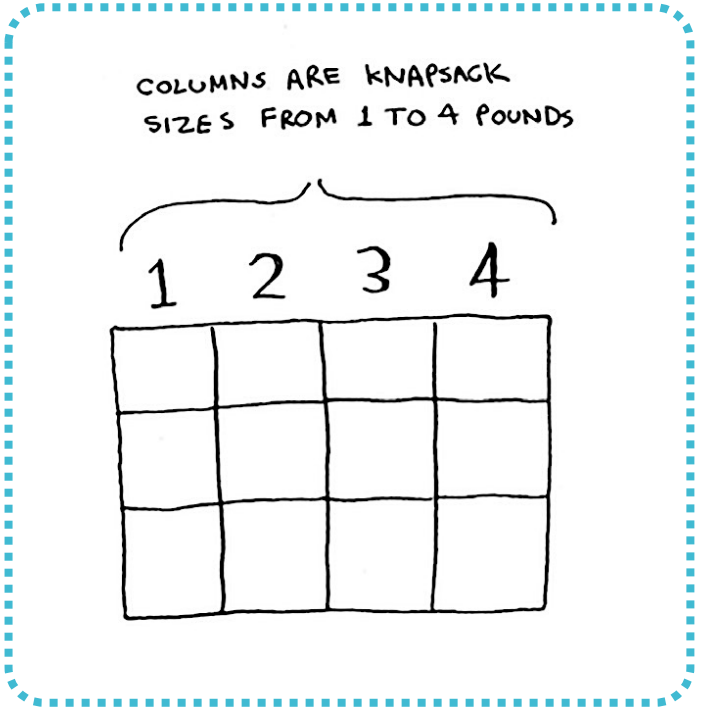
- Similar to Fibonacci problem, again, **recursion is too slow**.
- Dynamic programming solve the problem more **efficiently**.

- Lets check another problem.



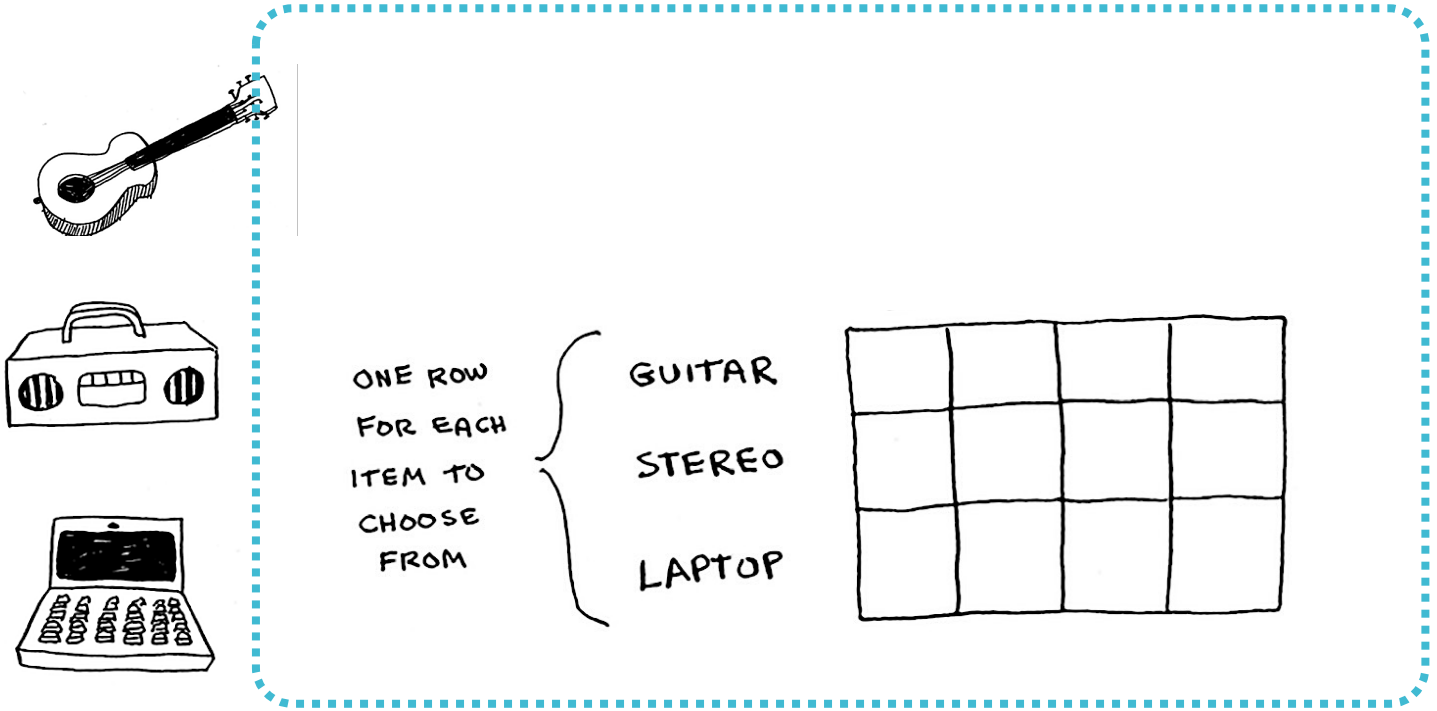
Dynamic Programming

- First, lets **begins with a table** (a grid).
  - **columns** are knapsack **capacity** from 1 - 4 lb.



Dynamic Programming

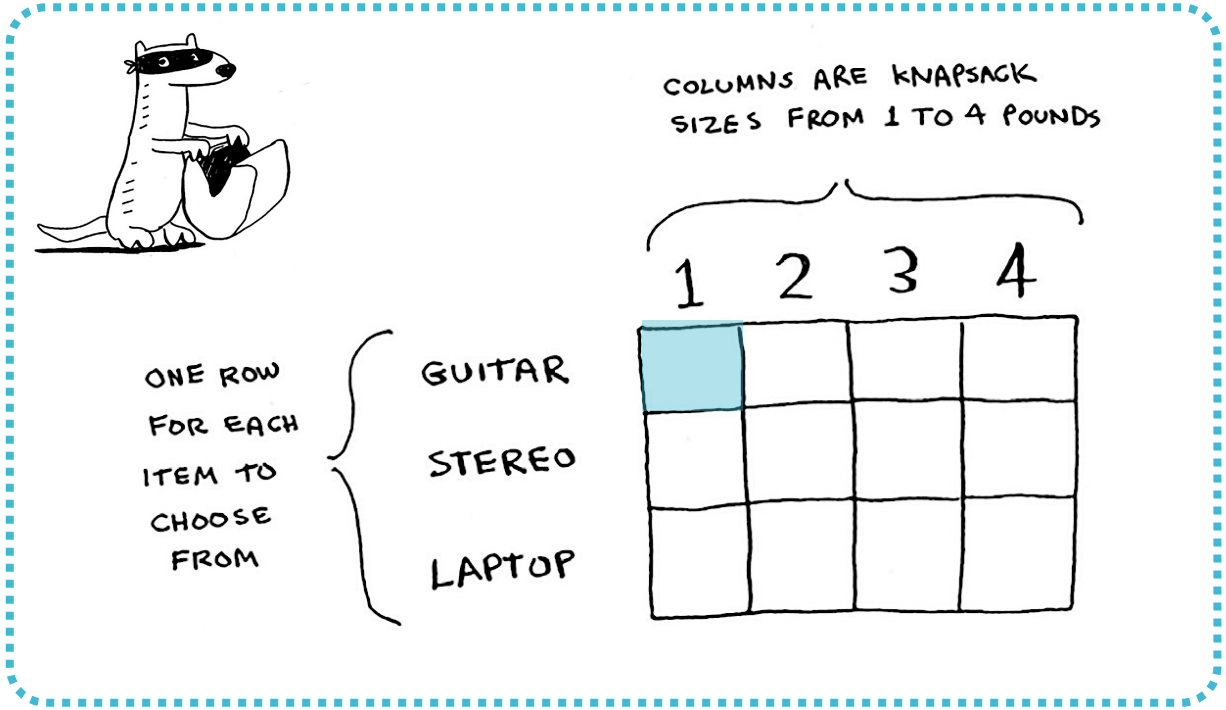
- First, lets **begins with a table** (a grid).
  - **rows** of the grid are the **items** to steal!



Dynamic Programming

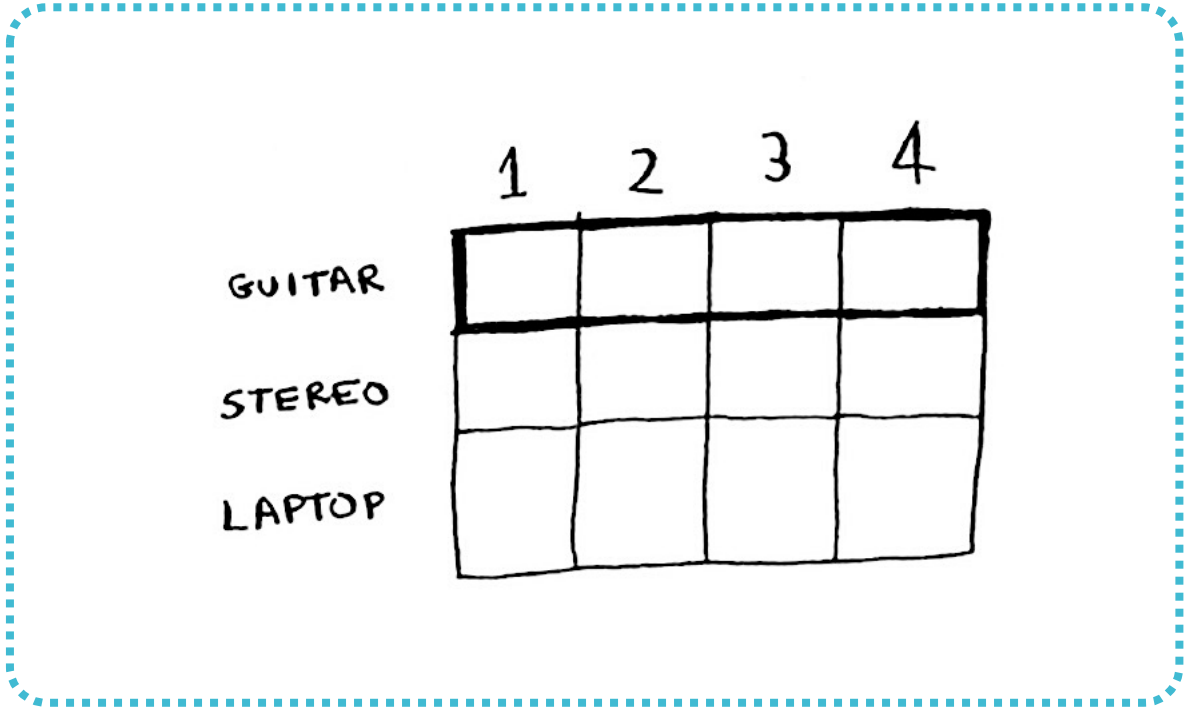
- The table (grid) will help us **to calculate and store** maximum value to put knapsacks.
- At each cell, there's a simple decision:

  *do we **put the item in the knapsack** or not?*



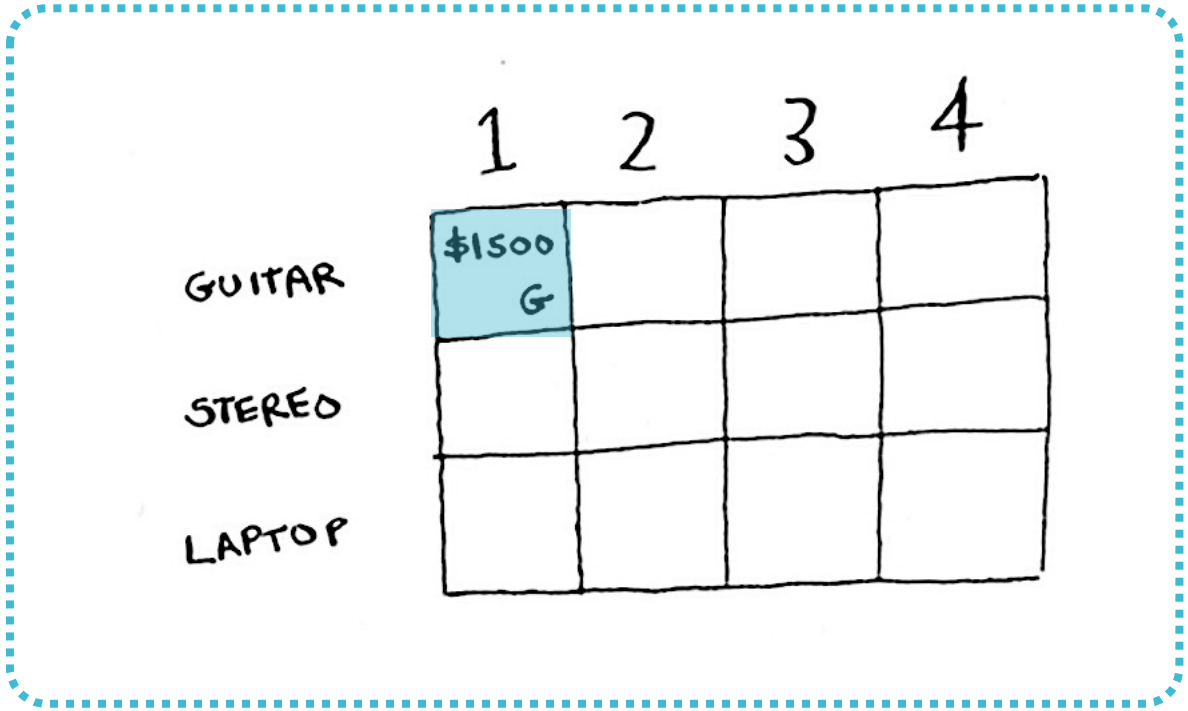
Dynamic Programming

- The table (grid) starts out **empty**.
- We fill in each cell of the table and **when the table is filled in**, we should find the answer!



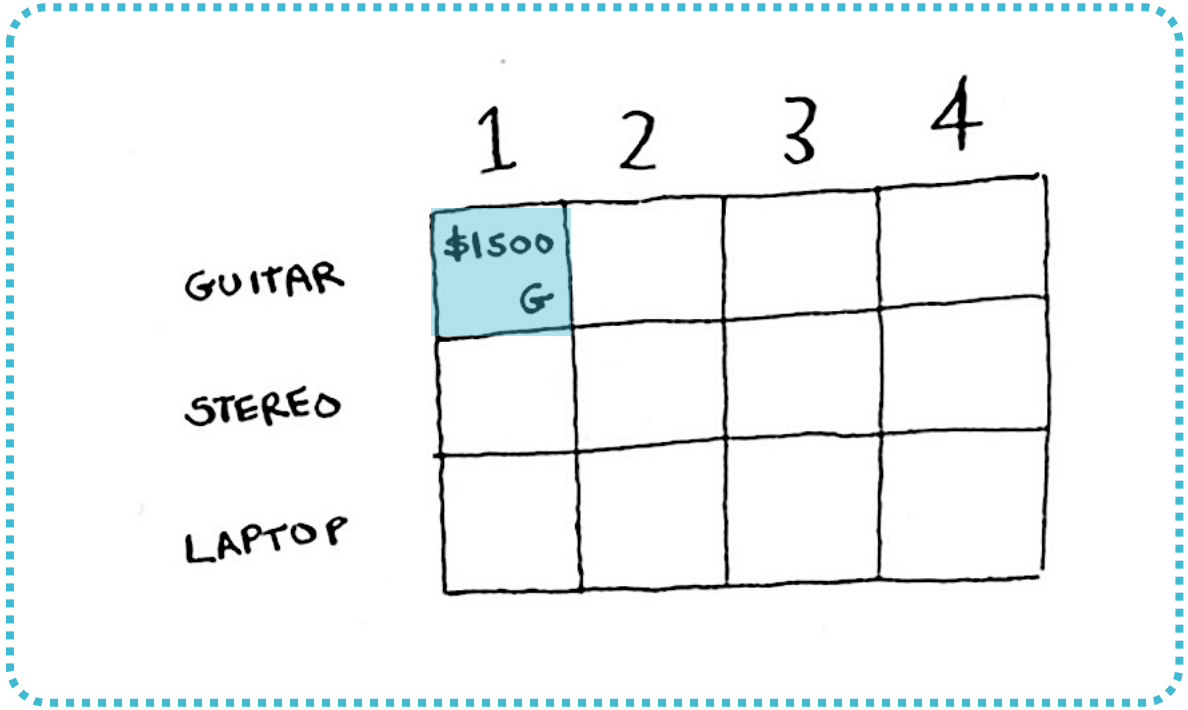
Dynamic Programming

- Let's start with the first row, **Guitar**.
- Do we put guitar **in the knapsack** and steal it?



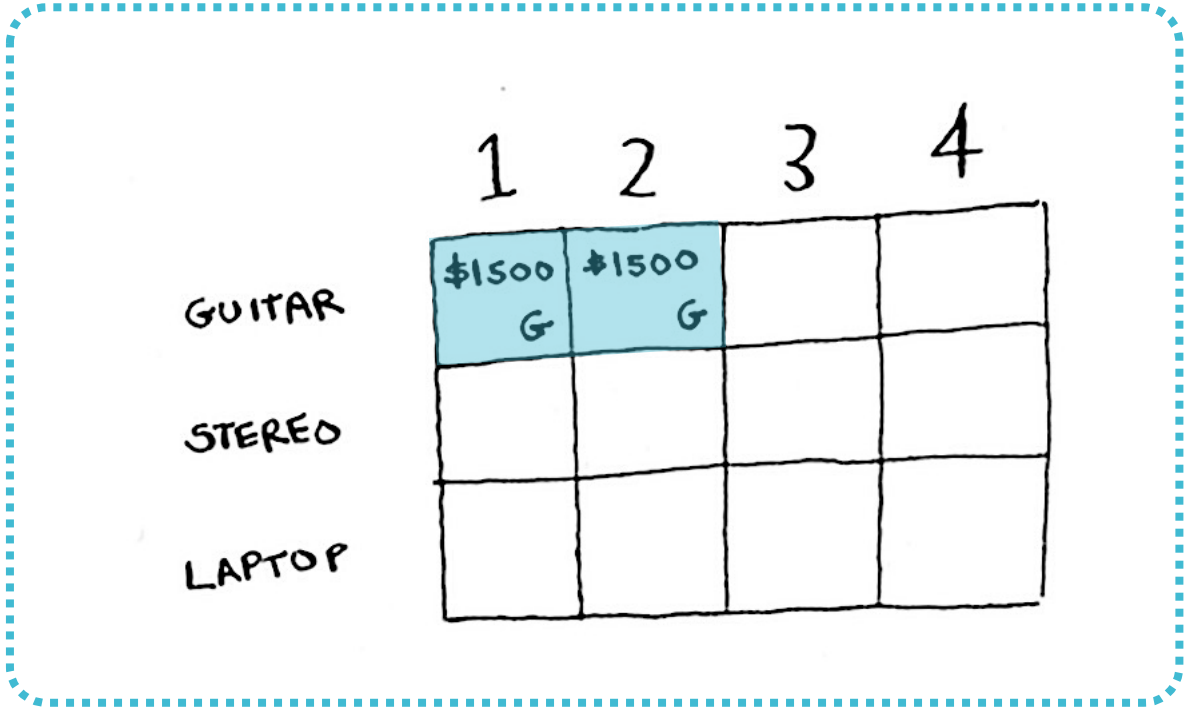
Dynamic Programming

- The first cell has a knapsack of **capacity 1 lb**.
- **The guitar is also 1 lb**, which means it fits into the knapsack! Its value is **$1500**.



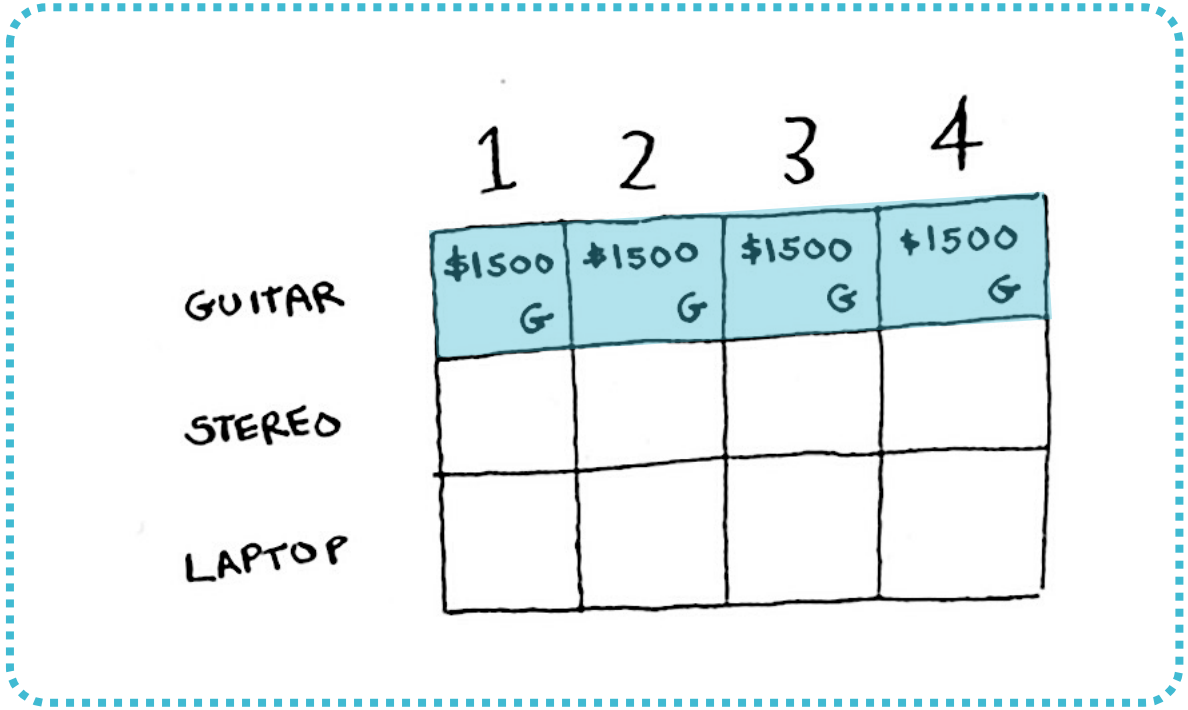
Dynamic Programming

- Like this, each cell in the grid will contain a list of all the items that fit into the knapsack.
- Next cell has **capacity 2 lb**: guitar definitely fits!



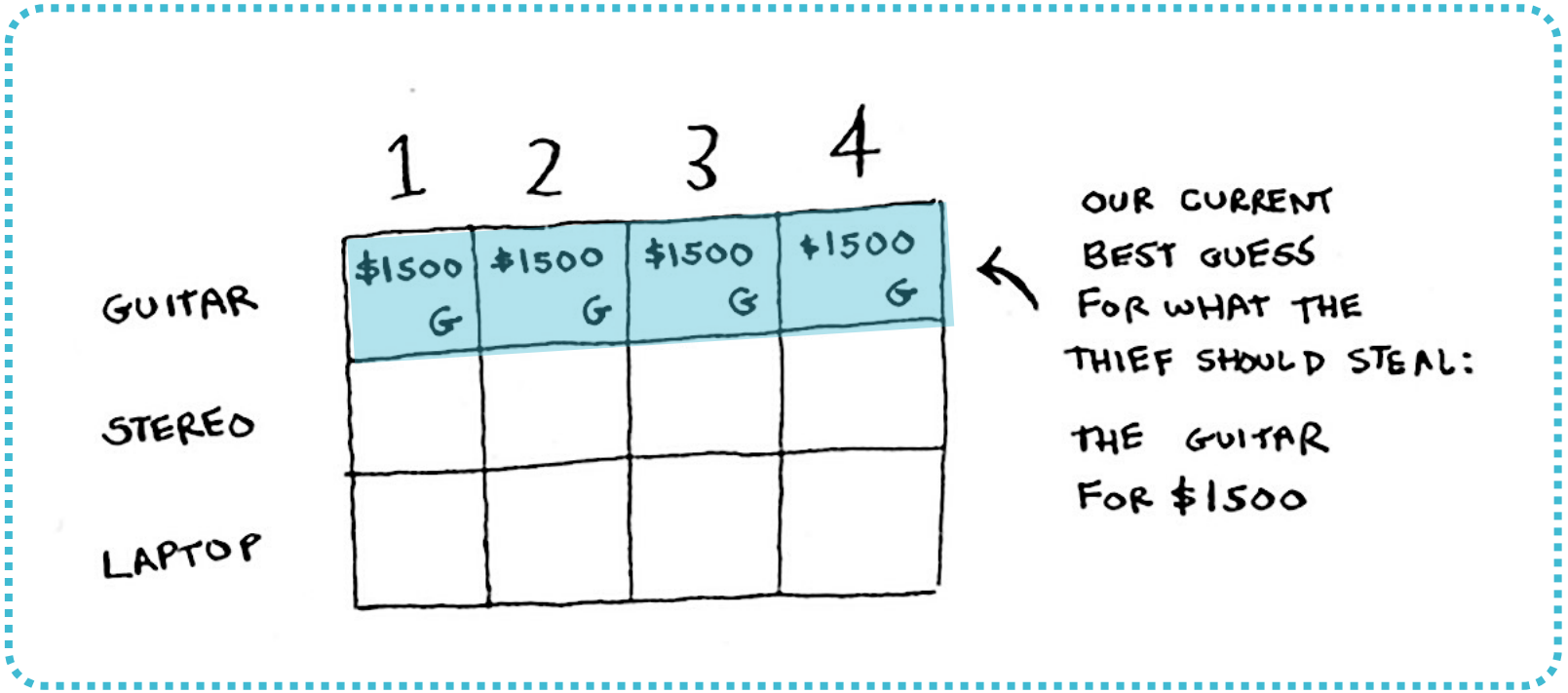
Dynamic Programming

- The same for the **rest of the cells** in this row.
- And **guitar** is the only item considered so far!



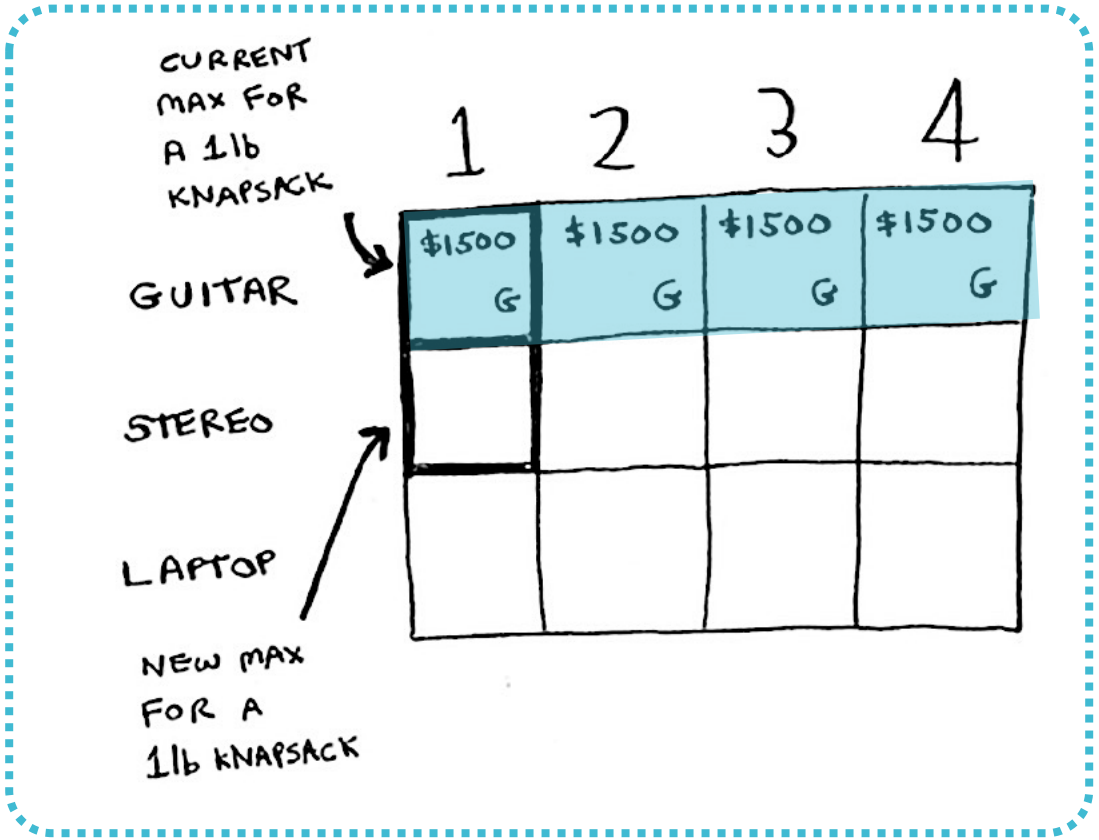
Dynamic Programming

- But why doing this for capacities of **1 lb** … **4 lb**?
- Well … , we are solving **smaller subproblems** that will help to solve the original bigger problem.



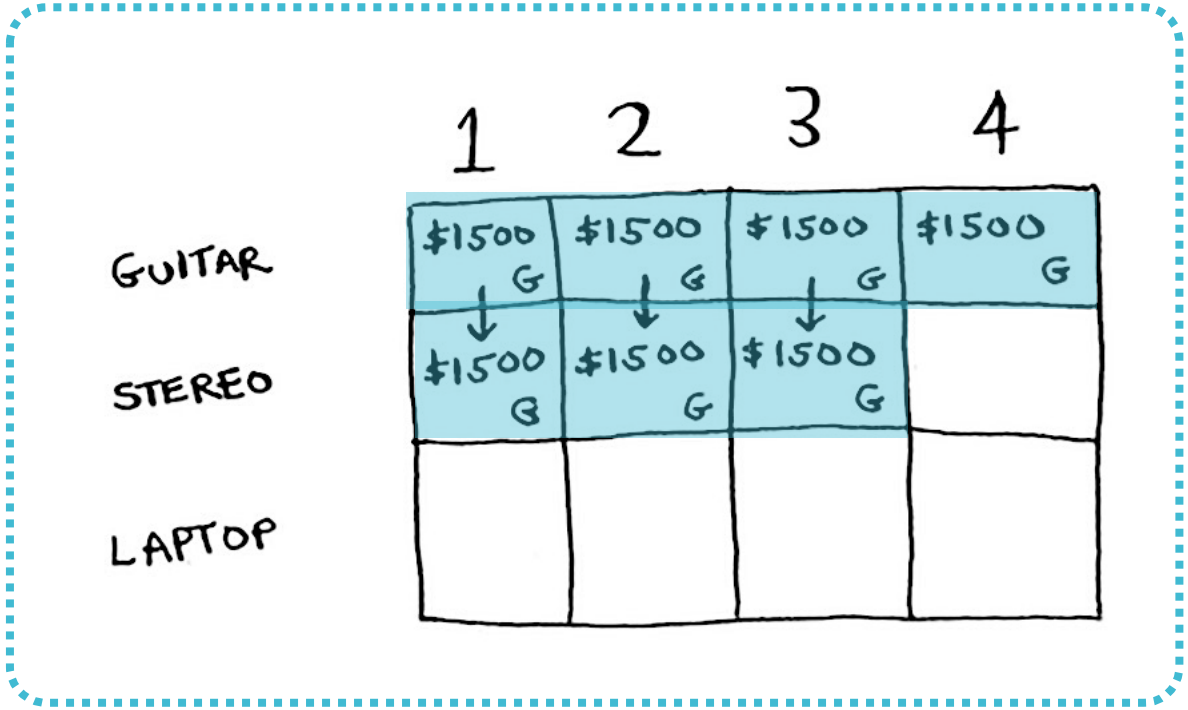
Dynamic Programming

- **At every row**, we can steal the item at that row or the items in the rows above it if they fit the knapsack.



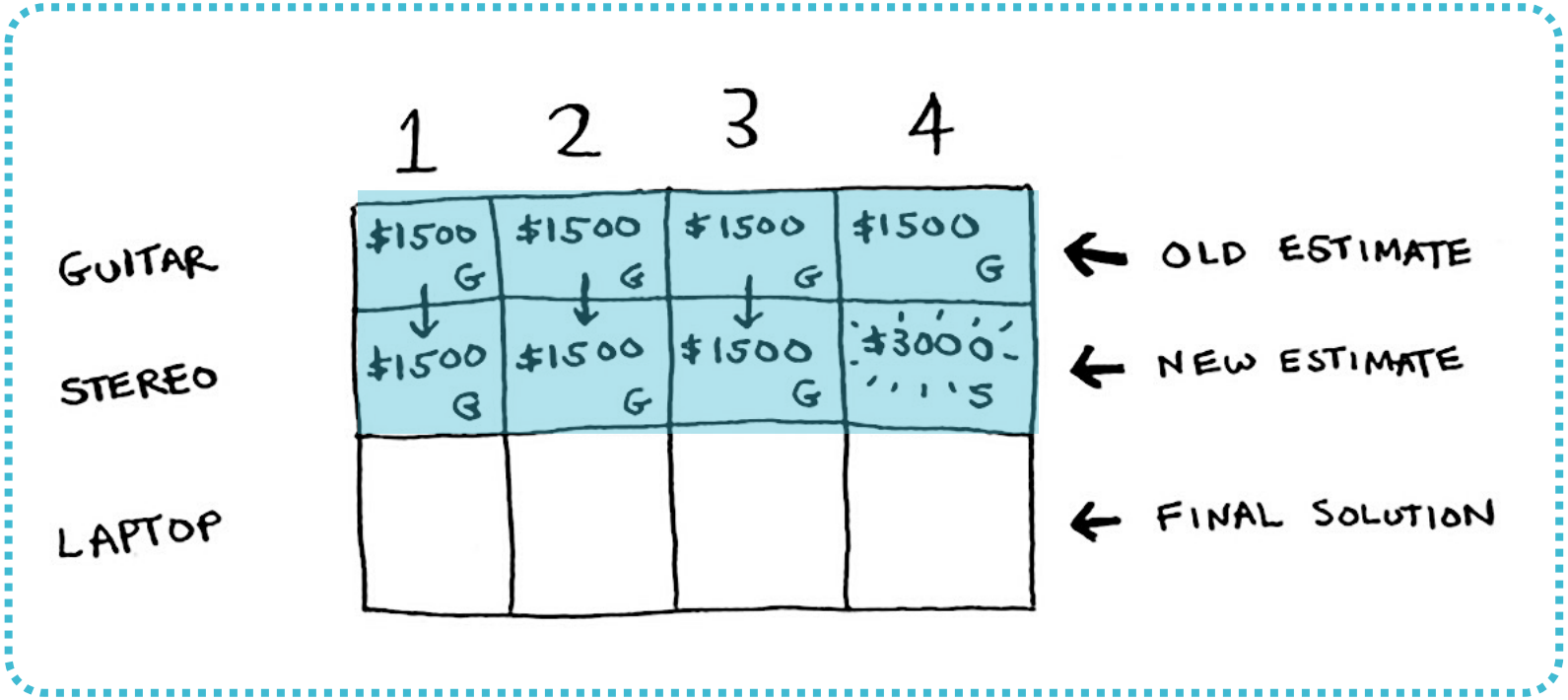
Dynamic Programming

- For the next two cells the **stereo doesn't fit**, so the item remain unchanged and **$1,500** remains the **max value**.



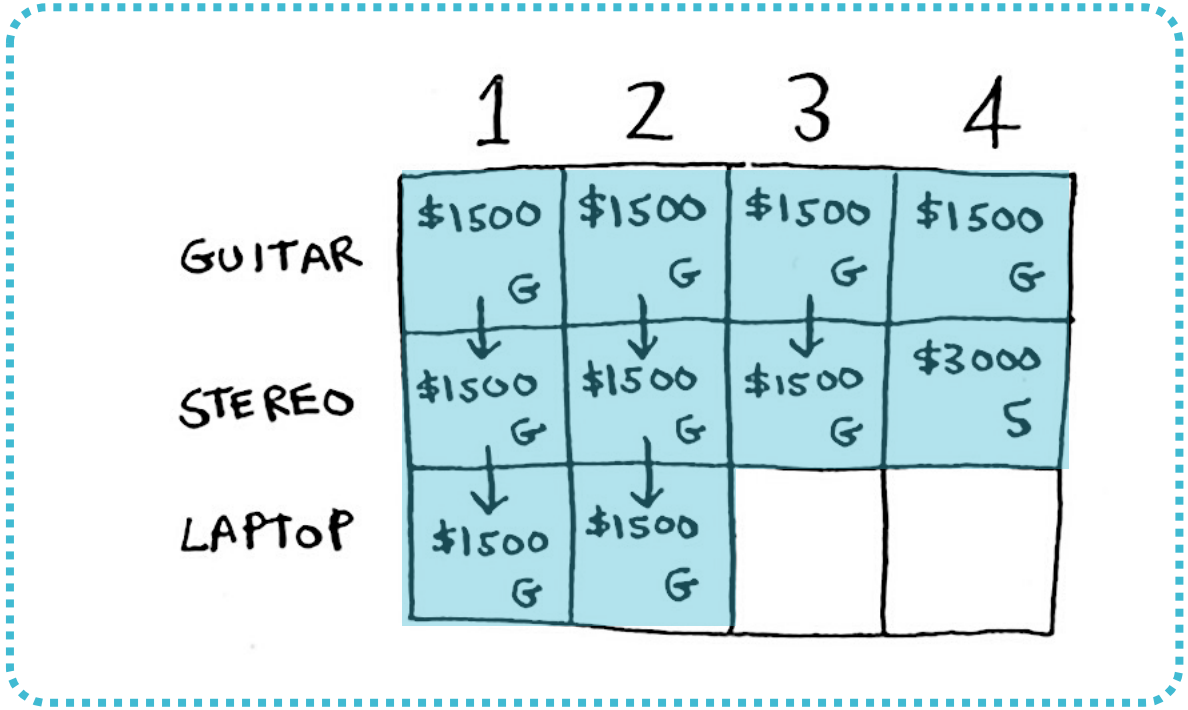
Dynamic Programming

- However, **stereo will fit** if the capacity is 4 lb!
- And the **new max value will be $3,000**!



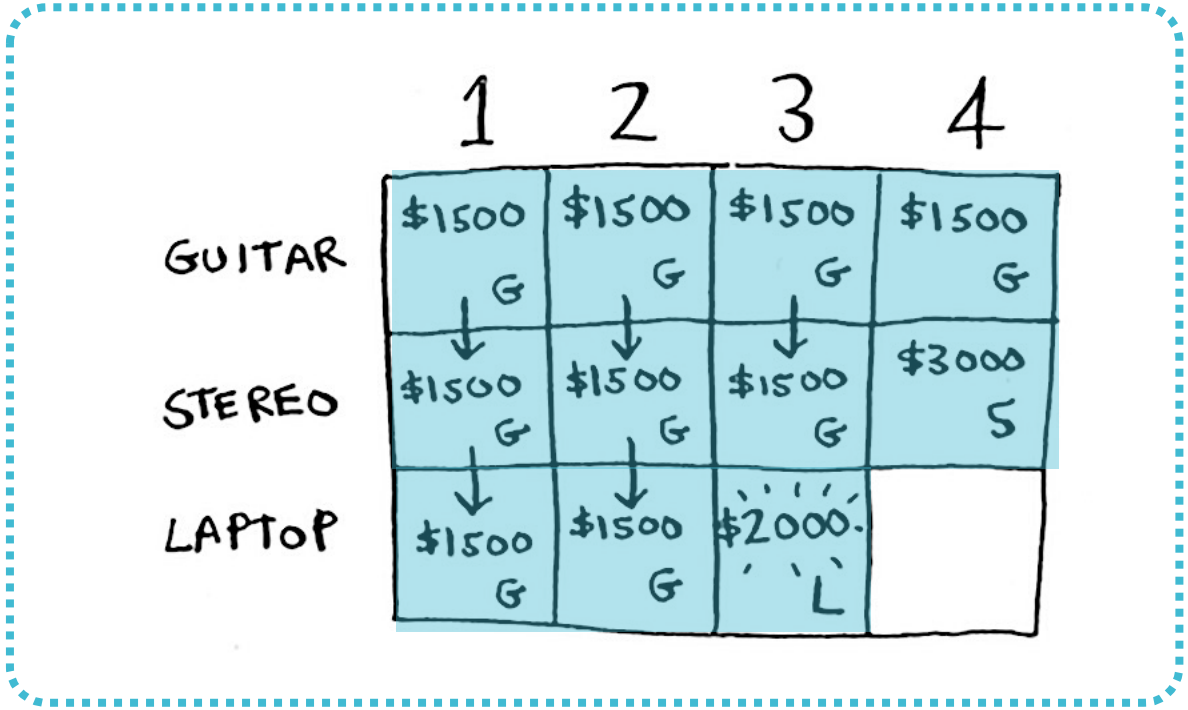
Dynamic Programming

- We just **continue with laptop**. It does not fit to the first cells (capacities 1-2 lb).



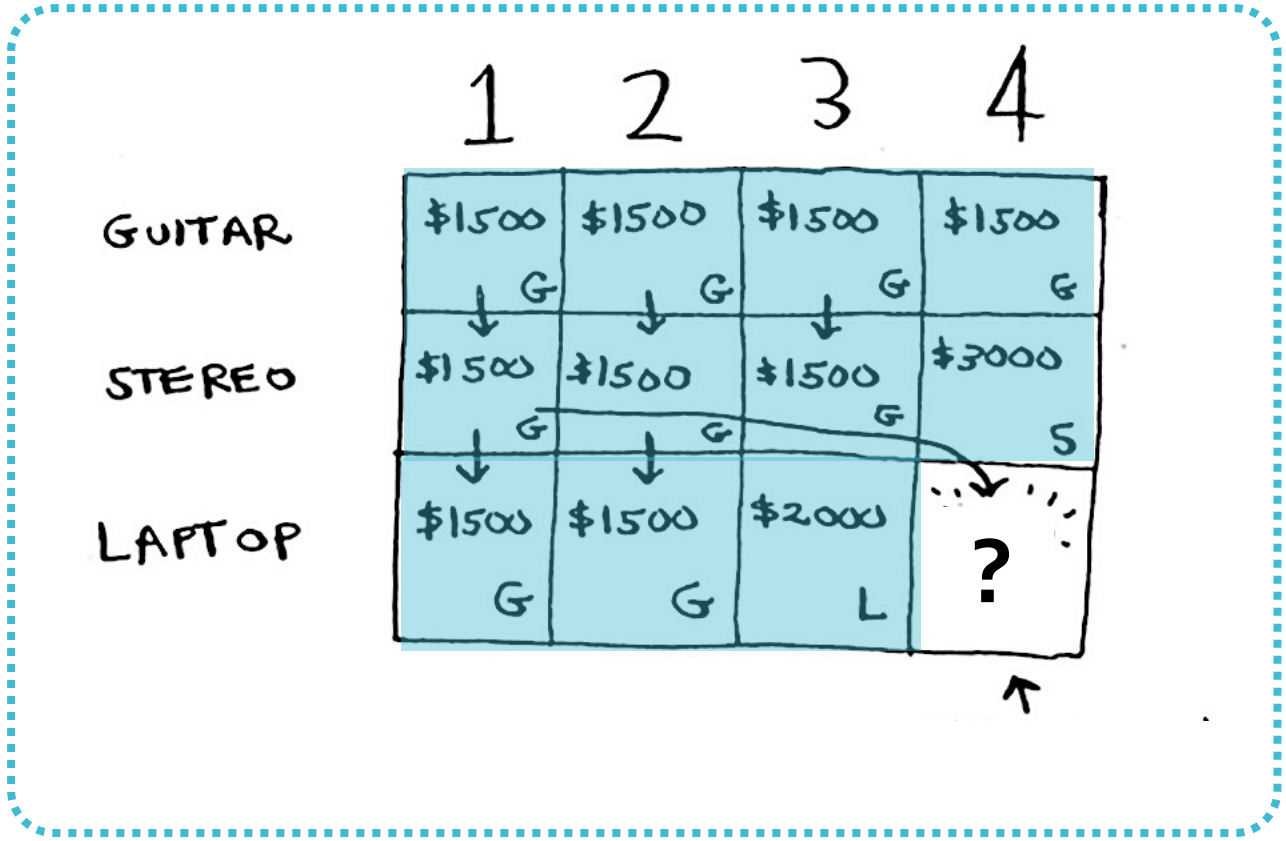
Dynamic Programming

- At the third cell (3 lb), **we can choose a laptop**, and then **new max will be $2,000**!



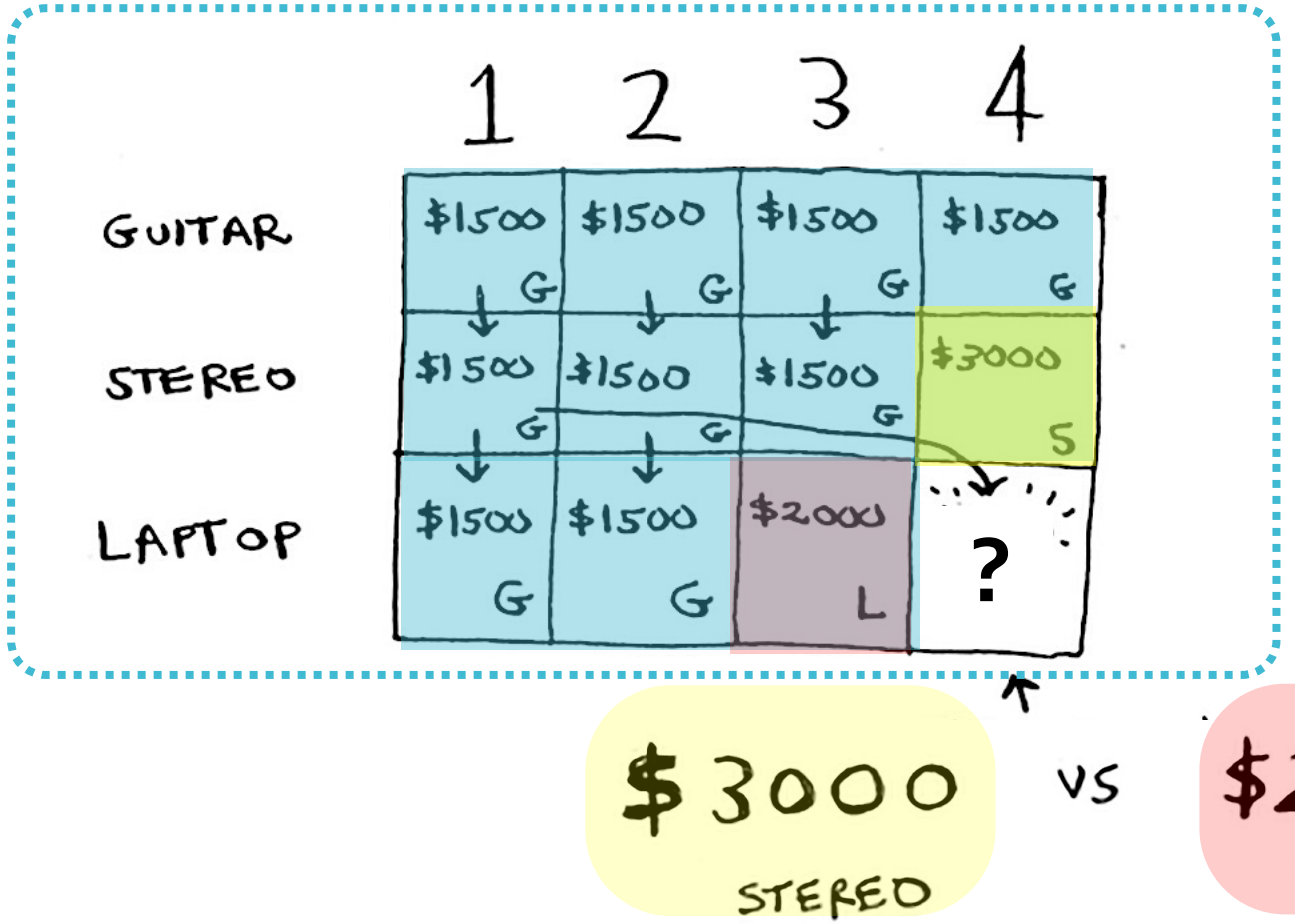
Dynamic Programming

- Now we have the **final cell** (all items & biggest capacity):



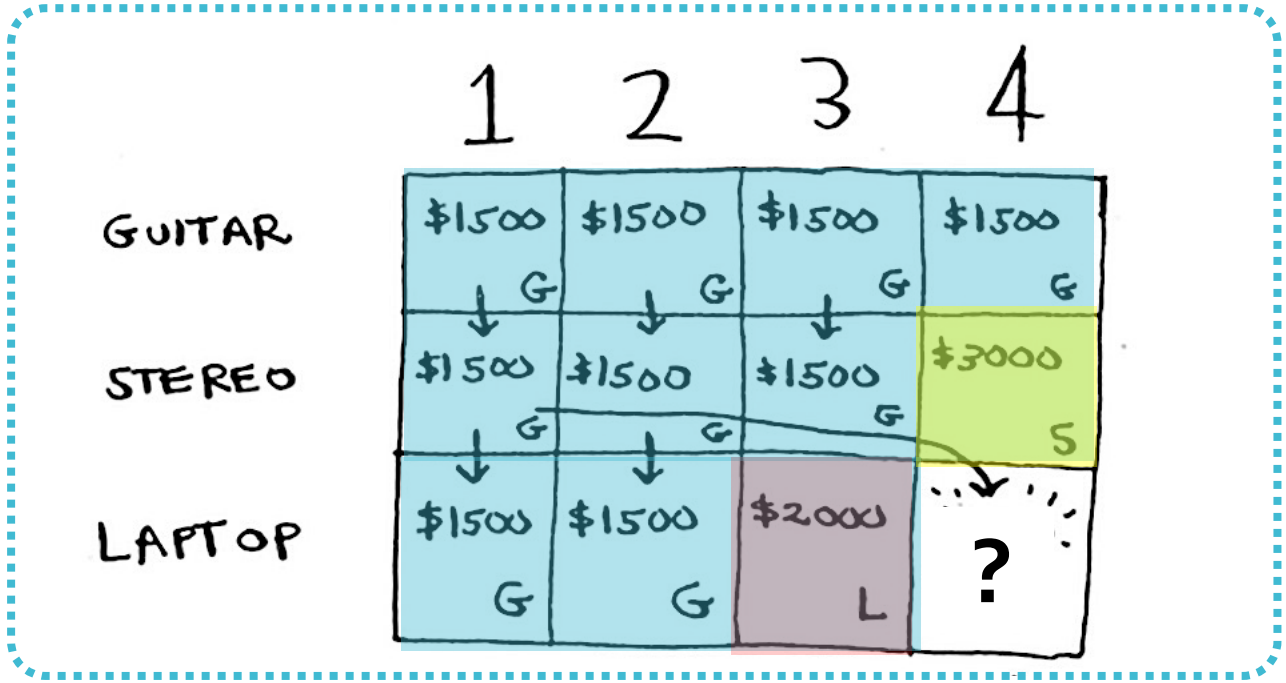
Dynamic Programming

- We can decide between two choices.
  - the above cell: **worth of $3,000** (stereo)
  - the left cell: **worth of $2,000** (laptop)



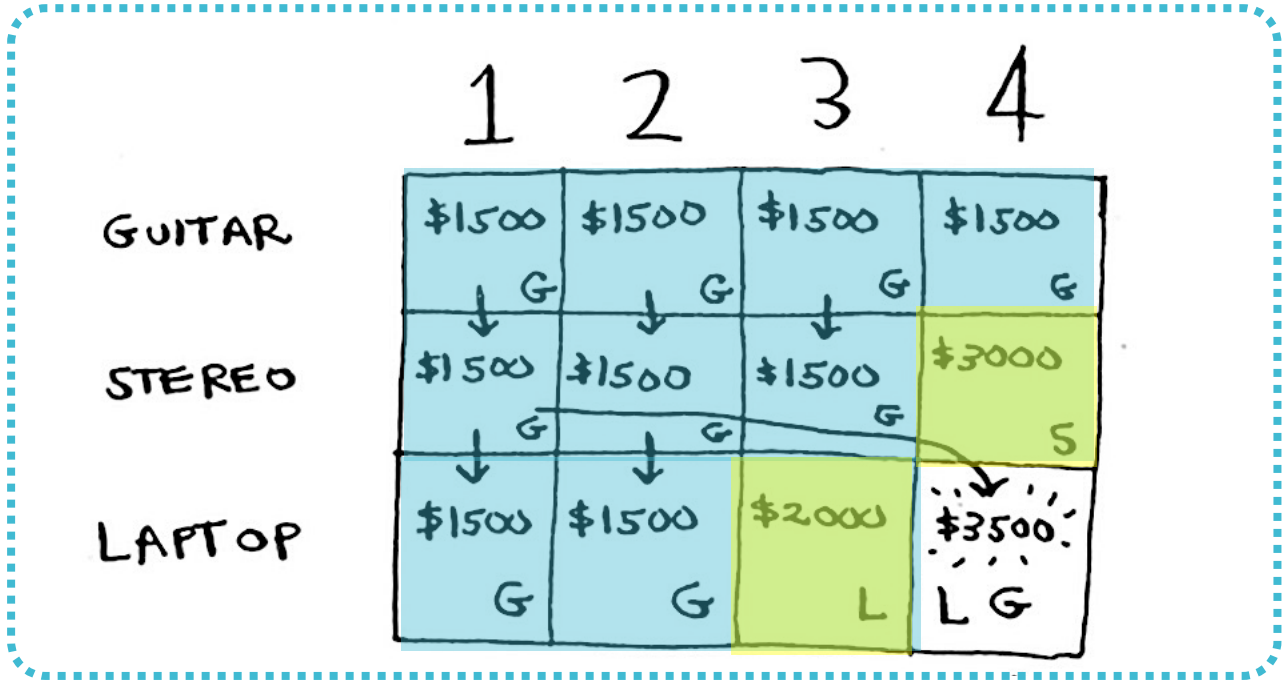
Dynamic Programming

- We can decide between two choices.
  - the above cell: **worth of $3,000** (stereo + **free space**)
  - the left cell: **worth of $2,000** (laptop)



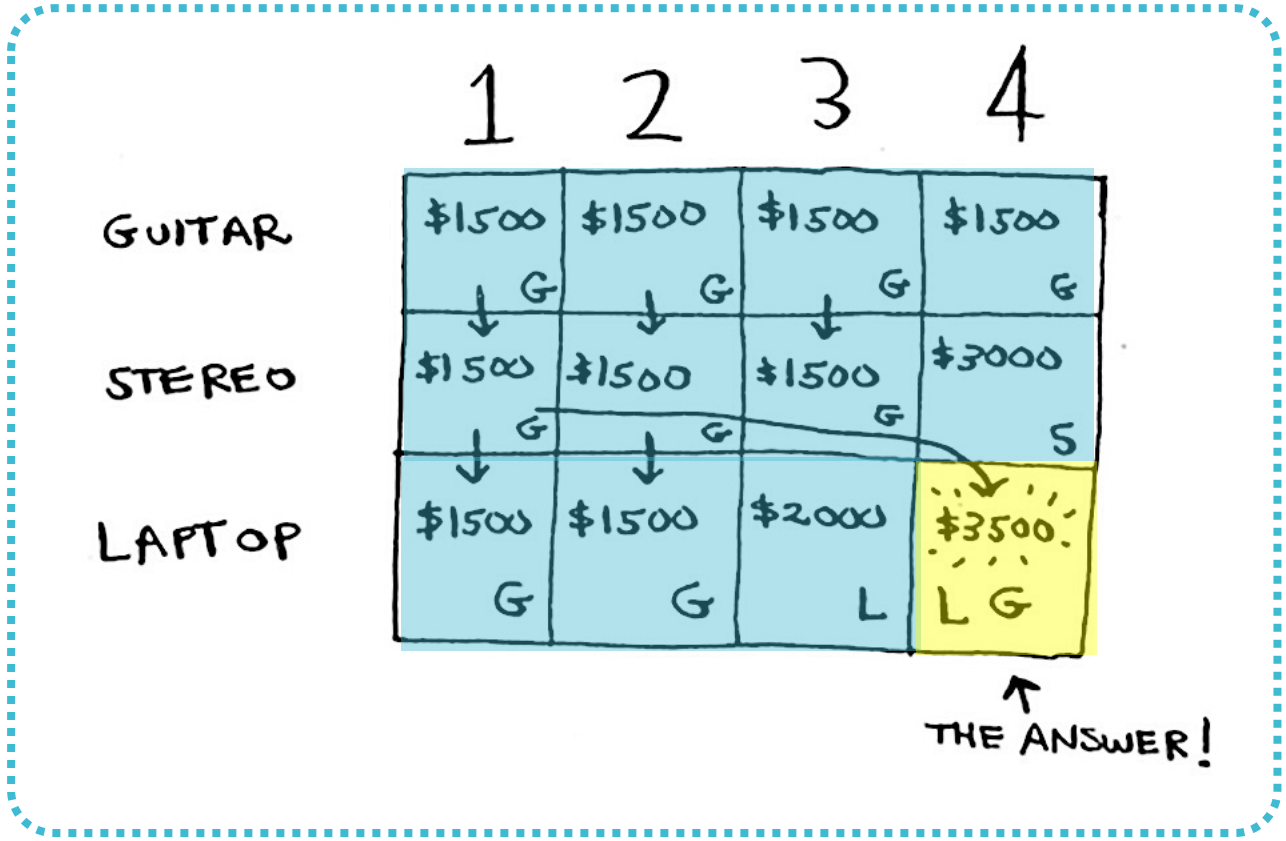
Dynamic Programming

- We can decide between two choices.
  - the above cell: **worth of $3,000** (stereo + **guitar**)
  - the left cell: **worth of $2,000** (laptop)



Dynamic Programming

- Calculating the max values for smaller knapsacks, that were **solutions for subproblem**, are helping us to find the final answer.



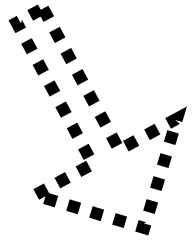
Dynamic Programming

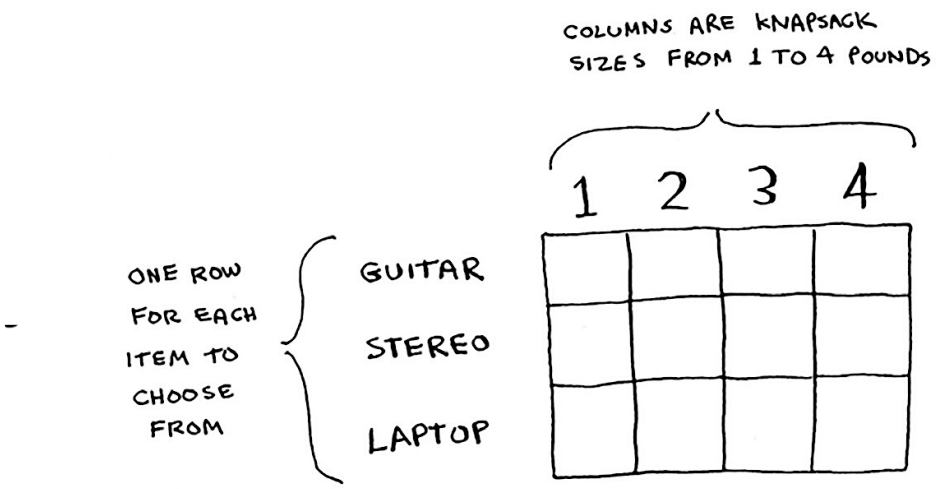
- This is called **Tabulation.**
- **Tabulation** is an approach to solve a problem by filling up a table (grid) with sub-solutions and using them to find a final solution.
- Tabulation is a **Bottom-Up** process of **storing sub-solutions**.

❖ Knapsack problem with **Dynamic Programming**

Dynamic Programming

```
def knapsack_dp(capacity, weights, values, n):
    grid = [[0 for x in range(capacity + 1)]
            for x in range(n + 1)]
```





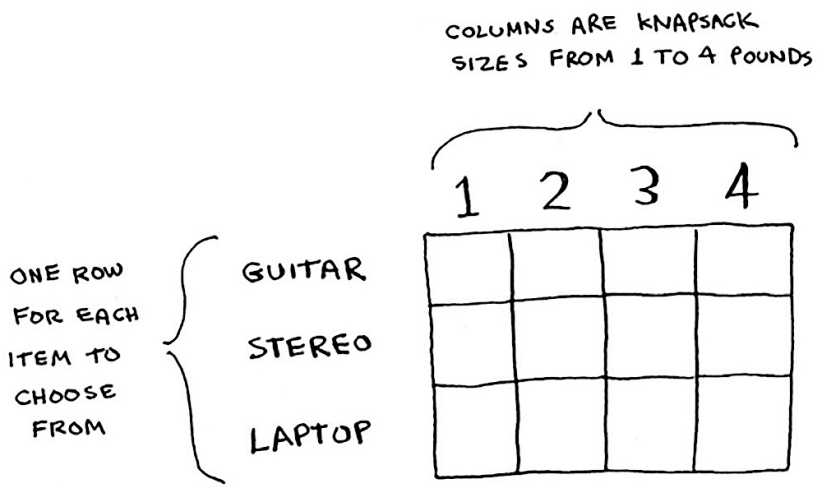
Dynamic Programming

Knapsack problem with **Dynamic Programming**

```
def knapsack_dp(capacity, weights, values, n):

    for item in range(n + 1):
        for cap in range(capacity + 1):
```

loop over items (rows)

loop over different capacity values (columns)



Dynamic Programming

Knapsack problem with **Dynamic Programming**

```
def knapsack_dp(capacity, weights, values, n):
```

```
            if item == 0 or cap == 0:
                grid[item][cap] = 0
```

Dynamic Programming

❖ Knapsack problem with **Dynamic Programming**

```
def knapsack_dp(capacity, weights, values, n):
```

```
            elif weights[item - 1] <= cap:
                grid[item][cap] = max(values[item - 1] +
                                      grid[item - 1][cap - weights[item - 1]],
                                      grid[item - 1][cap])
```

Dynamic Programming

❖ Knapsack problem with **Dynamic Programming**

```
def knapsack_dp(capacity, weights, values, n):
```

```
                else:
                    grid[item][cap] = grid[item - 1][cap]
```

Dynamic Programming

❖ Knapsack problem with **Dynamic Programming**

```
def knapsack_dp(capacity, weights, values, n):
```

```
    return grid[n][capacity]
```

Knapsack problem with **Dynamic Programming**

Dynamic Programming

```
def knapsack_dp(capacity, weights, values, n):
    grid = [[0 for x in range(capacity + 1)]
            for x in range(n + 1)]

    for item in range(n + 1):
        for cap in range(capacity + 1):

            if item == 0 or cap == 0:
                grid[item][cap] = 0

            elif weights[item - 1] <= cap:
                grid[item][cap] = max(values[item - 1] +
                                      grid[item - 1][cap - weights[item - 1]],
                                      grid[item - 1][cap])

            else:
                grid[item][cap] = grid[item - 1][cap]

    return grid[n][capacity]
```

Dynamic Programming

Testing our implementation

```
item_val = [3000, 2000, 1500]

item_wt = [4, 3, 1]
total_cap = 4
n_items = len(item_val)

print('Max value to put in knapsack of capacity W:')
print(knapsack_dp(total_cap, item_wt, item_val, n_items), '$')
```

prices [$]

weights [lb]

[Output:]

```
Max value to put in knapsack of capacity W:
3500 $
```

Try it at home!

- Use the implementation and calculate the max value of items (in NOK) that can be stollen in **knapsack with 64 Kg capacity** with these items:

  - **Item 1** is 8 [Kg] and worth 50 [NOK]
  - **Item 2** is 16 [Kg] and worth 100 [NOK]
  - **Item 3** is 32 [Kg] and worth 150 [NOK]
  - **Item 4** is 40 [Kg] and worth 200 [NOK]

Comparison

Greedy Algorithms	Divide and Conquare	Dynamic Programming
Optimizing by making the best choice at the moment.	optimizes by breaking down the problem into subproblems and using recursion to solve	optimizes by caching the answers to each subproblem as not to repeat the calculation.
Does not find always the optimal solution but it is fast.	Finds optimal solution but it is slower than Greedy Algorithms.	Finds optimal solution but may be pointless on small data.
May require almost no memory.	May require some memory.	May require a lot of memory.
Example: Dijkstra's Algorithm	Example: Merge Sort	Example: memoized fibonacci

Ref: Skerritt.blog

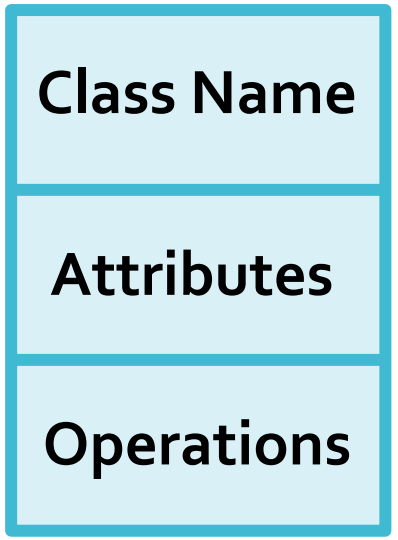
Object Oriented Programming

- **Object Oriented Programming (OOP)**

UML Diagrams

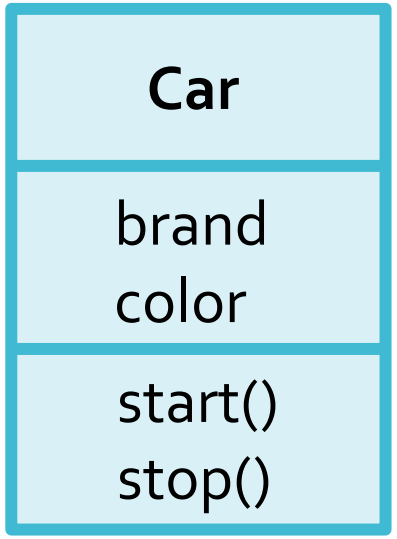
- **Unified Modeling Language (UML)** diagrams are used to show relationships among classes & objects.

- UML class diagram consists of:
  - Class Name
  - Attributes
  - Operations

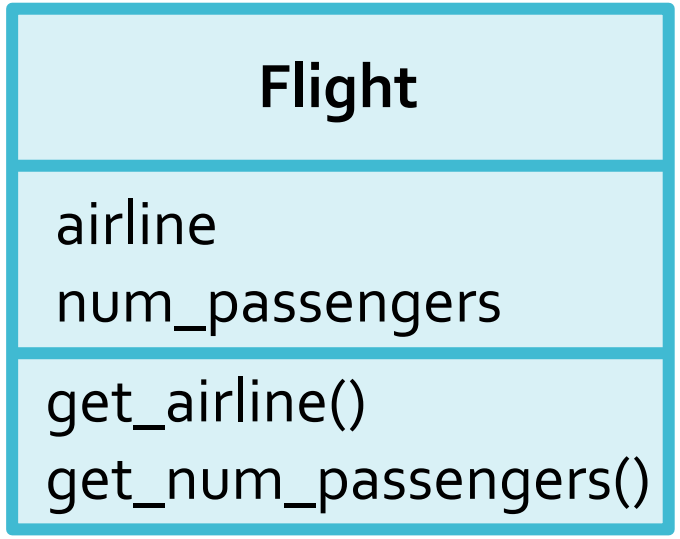


UML Diagrams

- **Example 1:**



- **Example 2:**



Quiz

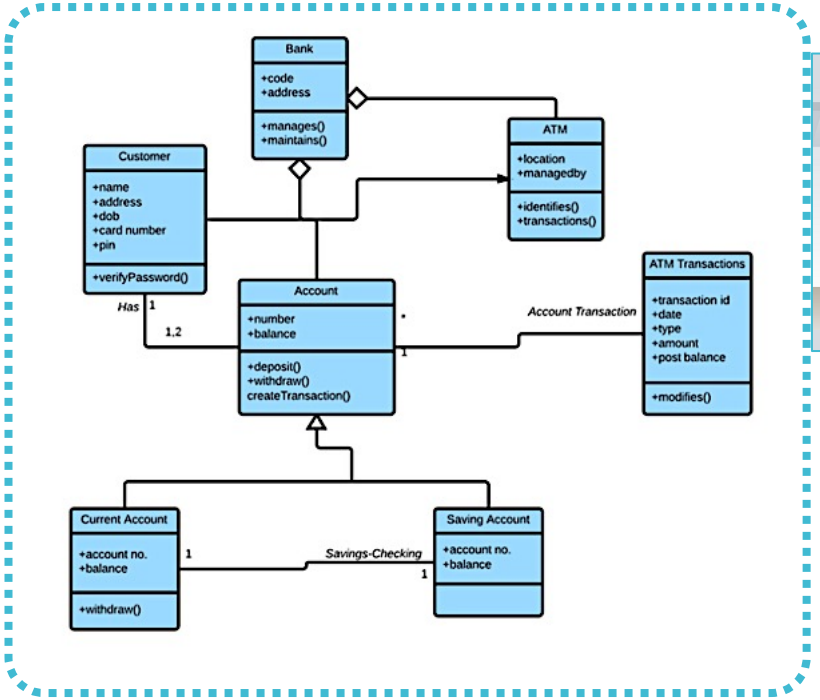
- You are already familiar with the concept of **class & object**.

- **Example:** which of these terms represent a class & which one an object?

a) Superhero, Superman
b) Sara, Person
c) Magazine, Economics
d) Easter, Holiday
e) Car, BMW
f) Lufthansa, Airline

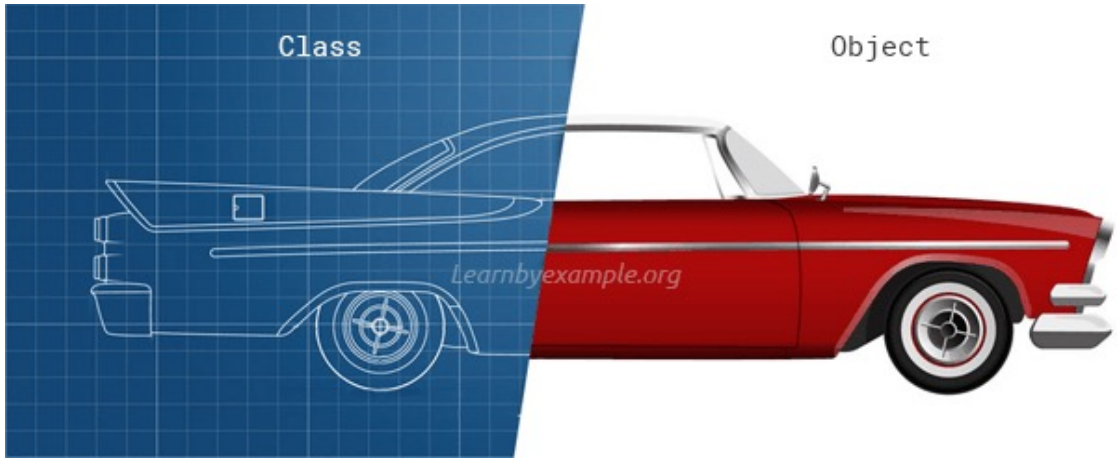
Object

- **Object-oriented:** a technique for modeling complex systems by describing a collection of interacting **objects** via their **data & behavior**.



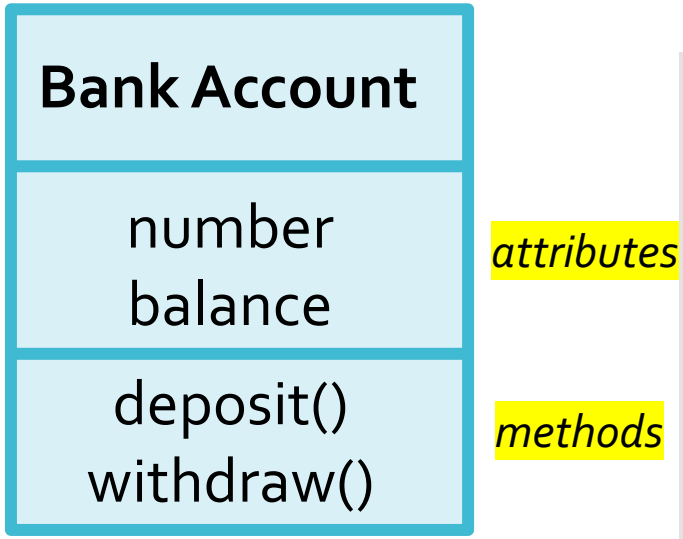
Class

- **Class:**
  - a blueprint for creating objects.
  - defines specific characteristics that are shared by all instances (objects) of that class.



Object Oriented Programming (OOP)

- A Python Class has:
  - **states** (attributes)
  - **behaviors** (methods)



- **Behaviours:** actions that can occur on an object.
- Behaviours that can be performed on a specific class of objects are called **methods**.

Quiz

- Methods are functions but what are the **differences** between **methods** and **functions** in Python?

- **Method**:

  ?

- **Function**:

  ?

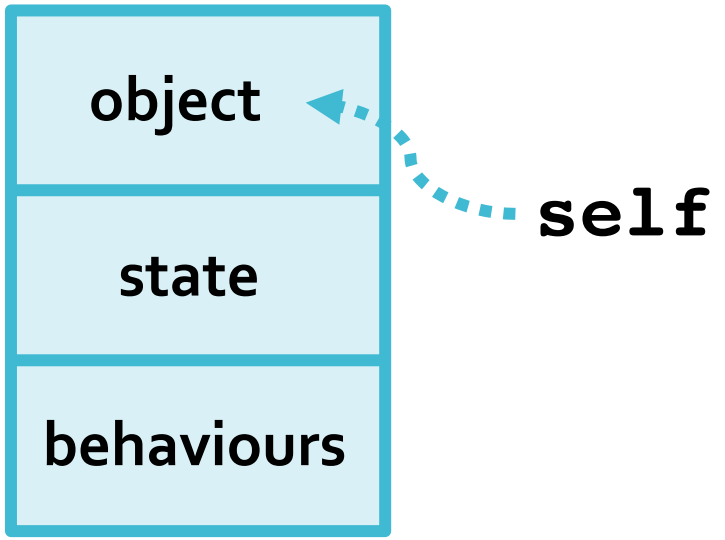
Object Oriented Programming

- To add an **attribute**, we create a variable and each object maintains its own copy of attributes.
- To add a **behaviour**, we create methods (which is a function defined within a class).

Object

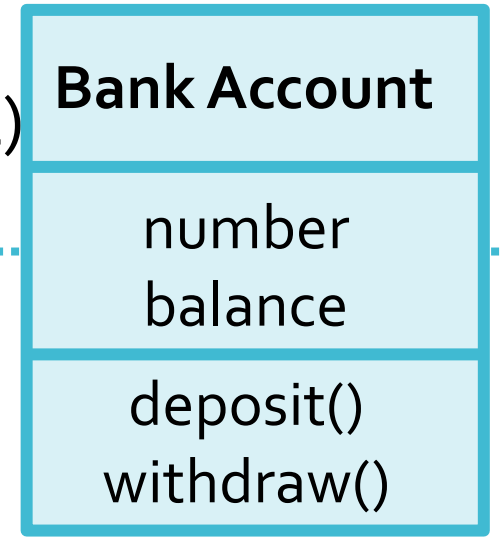
`__init__:` initialization method which is similar to constructors, used to initialize the state of the object.

`self:` a reference to the object that the method is being invoked on.



Example

❖ **Example:** Bank Account Class (p1)



attributes

methods

```
class BankAccount:

    def __init__(self, number):
        self.number = number
        self.balance = 0

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

More Examples

❖ Example:

Class	State (attributes)	Behaviour (methods)
Car	model year price speed	start() brake() accelerate() get_price()
Student	name id address grade	set_address() get_address() set_grade() get_grade()
Flight	origin destination airline number	set_airline() get_airline() set_origin() get_origin()
Bank Account	name number balance type	set_name() get_name() set_balance() get_balance()

Object Oriented

- **Object Oriented Analysis (OOA)**
- **Object Oriented Design (OOD)**
- **Object Oriented Programming (OOP)**

Object Oriented Analysis

- **Object-oriented Analysis (OOA):** process of looking at a problem and turn it into a computer program, by identifying the objects and interactions between those objects.
- Analysis process is about **what needs to be done.**
- Output of the analysis is a set of **requirements.**

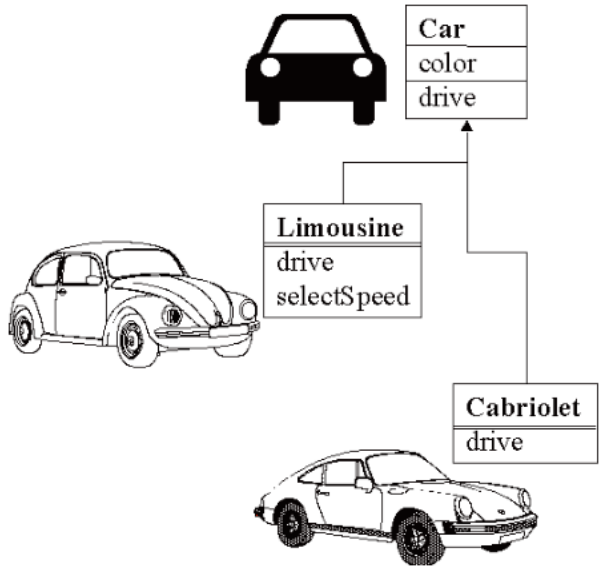
Object Oriented Analysis

- **Example:** Object-oriented Analysis (OOA) would turn a task, such as, "*A person needs a car*", into a set of requirements, such as:

- *Person* needs to:
  - *turnOn* the **engine** of the **car**
  - *accelerate* or *reduce* the **speed** of the **car**
  - *drive* the **car**

Object Oriented Design

- **Object-oriented Design (OOD):** process of converting requirements into implementation specifications, by:
  - naming the objects and defining the behaviors
  - specifying what objects activate other objects
- Design process is about **how things should be done.**
- **Output of the design** process is an implementation **specifications**.



Object Oriented Design

- **Example:** Object-oriented Design (OOD) would turn the requirements into a set of classes & interfaces.
- And they get implemented by an object-oriented **programming language** in the next phase.
- In this stage, programmer typically writes the **pseudocode** of the algorithm (or a flowchart of the algorithm).

Object Oriented Design

- **Pseudocode:** a description of an algorithm written in an English-like way rather than in a specific programming language.
- Pseudocode is useful since:
  - it can describe how an **algorithm must work**
  - It can explain a computing process **independent of the programming language**

```
procedure MatrixMultiplication(A, B)
  input A, B n*n matrix
  output C, n*n matrix

begin
  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      C[i,j] = 0;
    end for
  end for

  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      for( k = 0; k < n; k++)
        C[i,j] = C[i,j] + A[i,k] * B[k,j]
      end for
    end for
  end for
end MatrixMultiplication
```

Object Oriented Design

- Object-oriented design specify **needed classes** & **objects**, and how they interact.

- **Important decisions** are made during **requirements** and **design** process and implementation focuses on coding details.

- Some beginner programmers may think **writing code** is the core of software development, but it is not really!

Object Oriented Design

- Main task of object-oriented design is **determining the classes & objects** that will solve a problem.
- One way to identify potential classes is to identify the **objects discussed in the requirements**
- **Objects** are generally **nouns**, and the **services** that an object provides are generally **verbs**

Example

- **Another example:** Company with a website
- **Objects** are generally **nouns.**

The **client** needs to login and then can search for a **product** by the description, including the product name, ID, and **category**. If the product **ID** does not match the product, the **website** should show an error **message** and store the occurred **error** in the **log file**.

Example

- **Another example:** Company with a website
- **Methods** that objects provide are generally **verbs!**

The client needs to ***login*** and then can ***search*** for a product by the description, including the product name, ID, and category. If the product ID does not ***match*** the product, the website should ***show*** an error message and ***store*** the occurred error in the log file.

Important

- We need to keep in mind that:
  - a method should be **relatively small**, so that it can be understood as a single entity.
  - a **large** method should be **decomposed** into several **smaller methods** as needed for clarity.

Important

- If a class is too **complicated**, we can decompose it to smaller classes to divide tasks.

- Often it is difficult to **decide** whether or not something should be represented **as a class.**

- For example, the **model of a Car** can be represented as:
  - a **string** variable: **`model`**
  - an **object** of a class: **`Model`**

Object Oriented Programming

- **Object-oriented Programming (OOP)** is the process of converting the perfectly-defined design into a working program.

- In the this phase, the programmer translates the **pseudocode into the actual code** of a programming language.

Object-Oriented

- Object-oriented software development can typically involve four steps to follow:

1) generating the **requirements**
2) making a **design**
3) **implementing** the code
4) **testing** the implementation
5) **maintenance** which is the process of modifying a software after it has been tested & delivered to the customer (e.g., by correcting faults and improving performance).

Quiz

1) Given the following requirements of a software, what can be possible **classes** and **methods**.

The user will be able to enter a keyword and search for a video. Results should be ordered by upload date of the video. The user can also search videos based on the hash tags, inserted by the other users.

2) Which of the following is usually the **lengthiest process** in software development?
   a) Design
   b) Implementation
   c) Testing
   d) Maintenance
   e) None of the above

Next Lesson

- **Object Oriented Programming**