Introducción a las ciencias de la computación
y programación en Python

Diccionarios y recursión

Rodrigo Chang

Banco de Guatemala

Rodrigo Chang <rrcp@banguat.gob.gt>

Abstract

"Talk is cheap. Show me the code."
— Linus Torvalds

Veremos un nuevo tipo compuesto en Python: los diccionarios.
Hablaremos del concepto de recursión y veremos sus aplicaciones.

Diccionarios - almacenar información

- Supongamos que queremos guardar información de estudiantes de esta forma:

```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [2.00, 6.0001, 20.002, 9.01]
```

- Utilizamos una lista diferente para cada elemento.
  - Cada lista debe ser del mismo largo.
- Cada índice se refiere a la información de la misma persona.

Diccionarios - consultar información

- Veamos esta función para obtener la información de `student`:

```
def get_grade(student, name_list, grade_list,
    course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

- Es **enmarañado** si se debe registrar mucha información diferente.
- Se deben mantener muchas listas.
- Siempre deben ser índices enteros.
- Se debe recordar actualizar múltiples listas.

Los diccionarios: una forma más limpia

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index by label element

Los diccionarios: una forma más limpia

- store pairs of data
  - key
  - value

empty dictionary

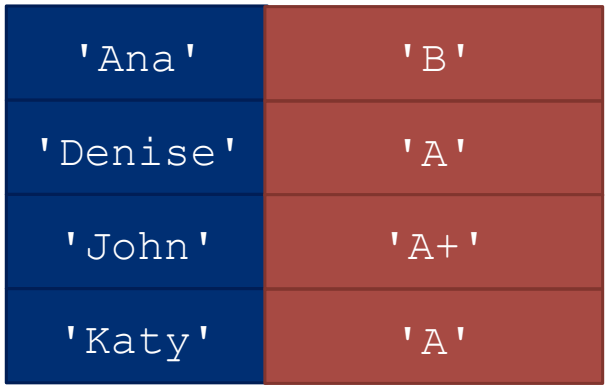
```
my_dict = {}
```

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

Los diccionarios: una forma más limpia

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error




```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

`grades['John']` → evaluates to `'A+'`

`grades['Sylvan']` → gives a `KeyError`

Operaciones de diccionarios

- `grades.keys()` devuelve un iterable que actúa como una tupla de todas las llaves.
- `grades.values()` devuelve un iterable que actúa como una tupla de los elementos almacenados.

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A
    '}

grades.keys() # ['Denise','Katy','John','Ana']

grades.values() # ['A', 'A', 'A+', 'B']
```

El orden no está determinado en estas funciones.

Llaves y valores

- Valores:
  - De cualquier tipo (mutables e inmutables).
  - Pueden ser duplicados.
  - Pueden ser listas o incluso otros diccionarios.
- Llaves:
  - Deben ser **únicas**.
  - Deben ser de tipos **inmutables**.
    - Cuidado con el tipo `float` utilizado como llave.
- **No hay un orden** en las llaves o valores:

```
d = {4:{1:0}, (1,3):"twelve", 'const'
    :[3.14,2.7,8.44]}
```

Listas vs. Diccionarios

- **Listas:**
- Secuencia ordenada de elementos.
- Búsqueda de elementos por índice entero.
- Los índices tienen orden.
- Índice o llave de tipo `int`.

- **Diccionarios:**
- **Asocian** una llave con un valor.
- Búsqueda de elementos a través de otros elementos.
- No existe un orden asociado.
- Llave o índice de tipo **inmutable**.

Ejemplo: letras de canciones

1. Crearemos un diccionario de frecuencias, es decir, una estructura de `str:int`
2. Encontraremos la palabra que más ocurre y cuántas veces.
   - Utilizamos una lista, por si hay más de una palabra.
   - Devolvemos una tupla `(list, int)`.
3. Encontraremos las palabras que ocurren al menos *X* veces.
   - Permitimos al usuario escoger *X* como parámetro.
   - Devolvemos una listsa de tuplas, donde cada tupla es `(list, int)`, conteniendo la lista de palabras ordenadas por frecuencia.
   - Del diccionario, obtenemos la palabra más frecuente y borramos la entrada. Repetimos. Esto funciona porque estamos mutando el diccionario de frecuencias.

Diccionario de frecuencias

```
def lyrics_to_frequencies(lyrics):
    myDict = {}
    for word in lyrics:
        if word in myDict:
            myDict[word] += 1
        else:
            myDict[word] = 1
    return myDict
```

Utilizando el diccionario

```
def most_common_words(freqs):
    values = freqs.values()
    best = max(freqs.values())
    words = []
    for k in freqs:
        if freqs[k] == best:
            words.append(k)
    return (words, best)
```

Aprovechando el uso de diccionario

```
def words_often(freqs, minTimes):
    result = []
    done = False
    while not done:
        temp = most_common_words(freqs)
        if temp[1] >= minTimes:
            result.append(temp)
            for w in temp[0]:
                #remove word from dict
                del(freqs[w])
        else:
            done = True
    return result
```

¿Qué es recursión?

> Es una forma en la cual se especifica un proceso basado en su propia definición.

- Algorítmicamente: una forma de utilizar divide y conquistarás.
  - Reducir un problema a versiones más simples del mismo problema.
- Semánticamente: técnica de programación donde una función **se llama a sí misma**.
  - La meta es **no** tener recursión infinita.
  - Deben existir **uno o más** casos base, simples de resolver.
  - Debe permitir resolver el mismo problema con otras entradas para **simplificar** el problema más grande inicial.

Algoritmos iterativos

- Las estructuras iterativas (ciclos `while` y `for`) llevan a los **algoritmos iterativos**.
- Capturan la computación en un conjunto de **variables de estado** que se actualizan en cada iteración del ciclo.

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

- El estado es capturado por la variable de iteración `i`.
- El valor actual de la computación es almacenado: `result = result + a`.

Algoritmos recursivos

```
def mult_recur(a, b):
    if b == 1:
        return a
    else:
        return a + mult_recur(a, b
    -1)
```

$$a * b = \underbrace{a + a + \ldots + a}_{b \text{ veces}}$$
$$= a + \underbrace{a + a + \ldots + a}_{b-1 \text{ veces}}$$
$$= a + a(b-1)$$

- **Paso recursivo:**
  - ¿Cómo reducir el problema a una versión más simple / pequeña del mismo problema?
- **Caso base:**
  - Seguir reduciendo el problema hasta obtener uno suficientemente simple que pueda ser resuelto directamente.
  - Cuando $b = 1$, $ab = a$

Factorial

- Recordemos que $n! = n * (n-1) * (n-2) * \ldots * 1$.

```
def factorial(n):
```

- **Caso base:** ¿para qué caso conocemos el factorial?

```
    if n == 1:
        return 1
```

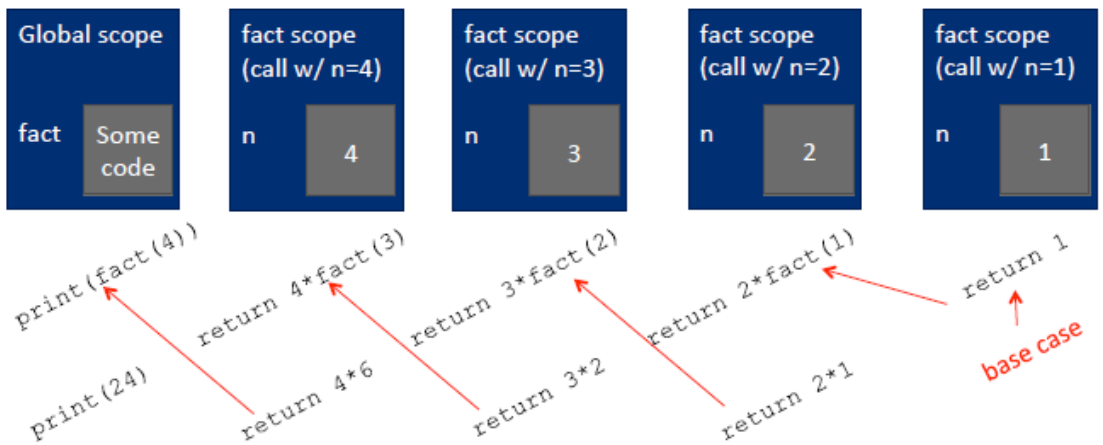
- **Paso recursivo:** escribimos el problema en términos de uno más simple para alcanzar el caso base.

```
    else:
        return n * factorial(n-1)
```

Factorial

- Ver en Python Tutor

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

print(fact(4))
```



Recursividad: observaciones

- Cada llamada a una función recursiva **crea su propio scope/ambiente**.
- Las asignaciones de variables en un ambiente no son afectadas por una llamada recursiva[1].
- El flujo de control regresa al **ambiente previo** cuando la llamada a la función retorna su valor.

[1] Al utilizar el mismo nombre de variable, pero son objetos diferentes en ambientes separados

Iteración vs. Recursión

```python
def factorial_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
```

```python
def factorial_rec(n):
    if n == 1:
        return 1
    else:
        return n *
    factorial_rec(n-1)
```

- La recursión puede ser más simple, más intuitiva.
- La recursión puede ser más eficiente desde el punto del programador.
- Puede no ser tan eficiente desde el punto de visa de la computadora.

Razonamiento inductivo

- ¿Cómo sabemos que nuestro código recursivo funcionará?
- `mult_iter` termina, porque `b` es inicialmente positivo y decrece en cada iteración, hasta que eventualmente `b < 0`.
- `mult` llamado con `b=1` es el caso base.
- `mult` llamada con `b>1` hace una llamada recursiva con `b` más pequeño ⇒ eventualmente se llamará con `b=1`.

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result

def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b
    -1)
```

Inducción matemática

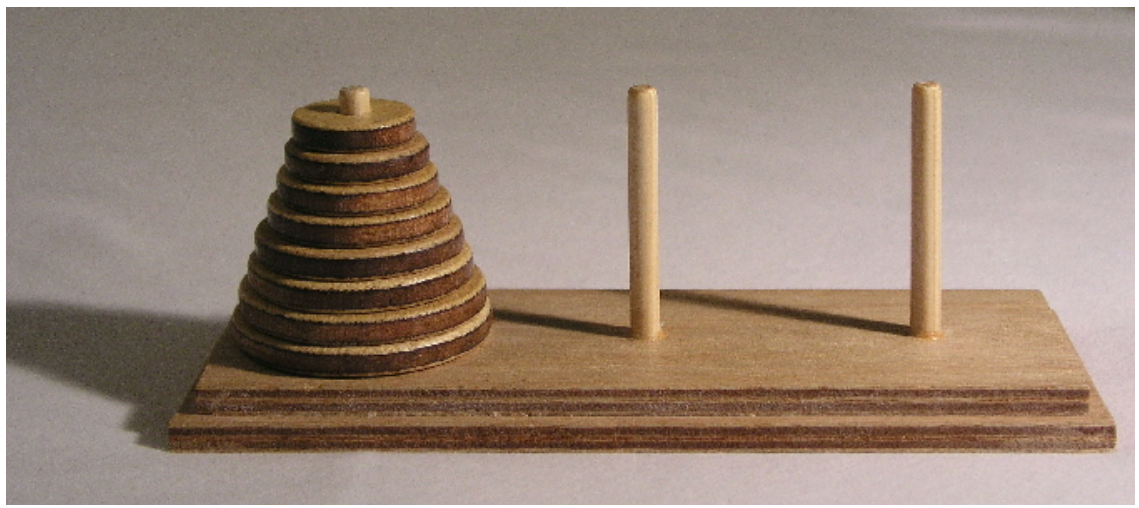
- Recordemos la **inducción matemática**: 1) asumimos P_1, luego probamos $P_n \Rightarrow P_{n+1}$.
- La misma lógica aplica:

```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

- Para el caso base, mostramos que `mult` devuelve un valor correcto.
- Para el caso recursivo, asumimos que `mult` retorna correctamente respuestas para problemas con más pequeños que `b`, por lo tanto, también lo hace para problemas de tamaño `b`.
- Por inducción, el código devuelve respuestas correctas.

Torres de Hanoi

- 3 postes verticales.
- Pila de discos de diferentes tamaños en uno de los postes.
- Se deben mover hacia otro poste. En este caso, ¡el universo termina!
- **Reglas:** solamente se puede mover 1 disco a la vez y un disco más grande no puede cubrir a un disco más pequeño.



Torres de Hanoi

- Habiendo visto algunos ejemplos, ¿cómo escribimos un programa para imprimir el conjunto de movimientos correctos?
- Debemos **¡pensar recursivamente!**
  - Resolver un problema más pequeño y más básico.

Torres de Hanoi

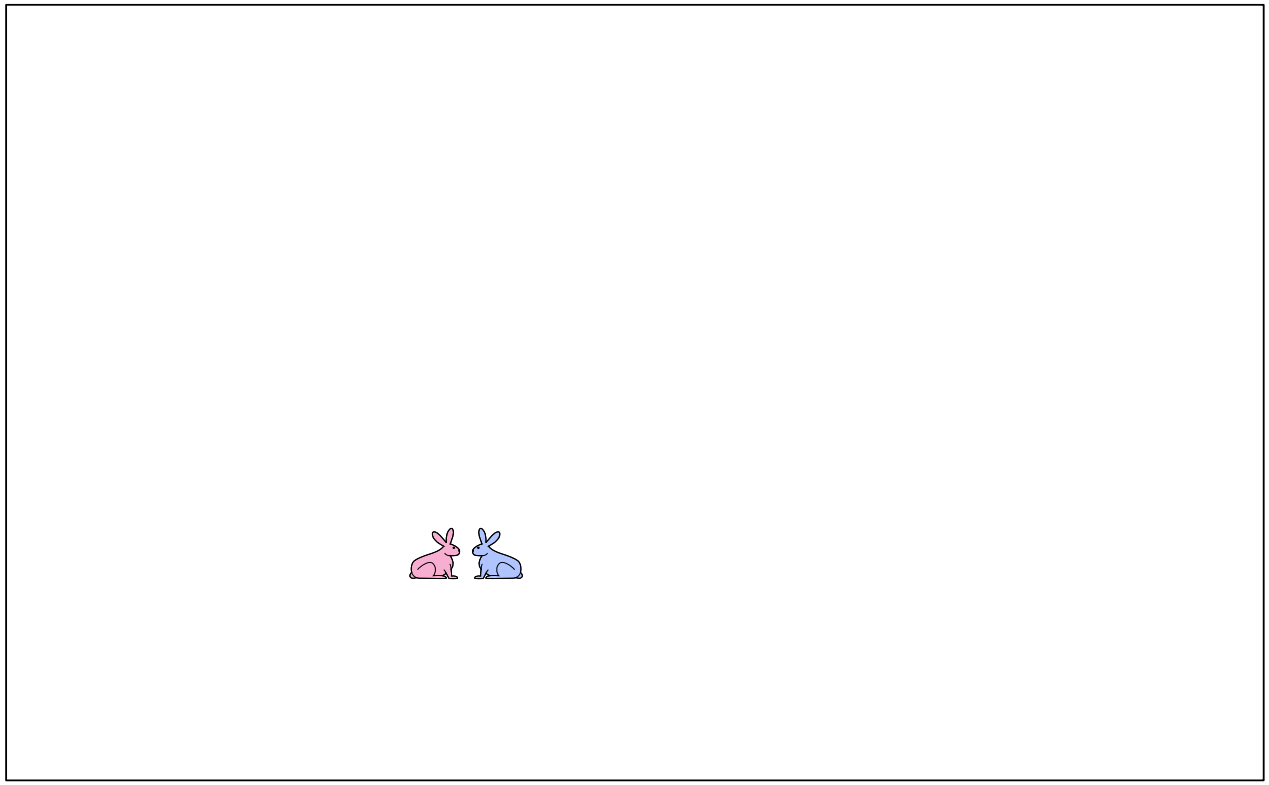
```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))

def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

Recursión con múltiples casos base

Leonardo de Pisa (Fibonacci) modeló el siguiente reto:

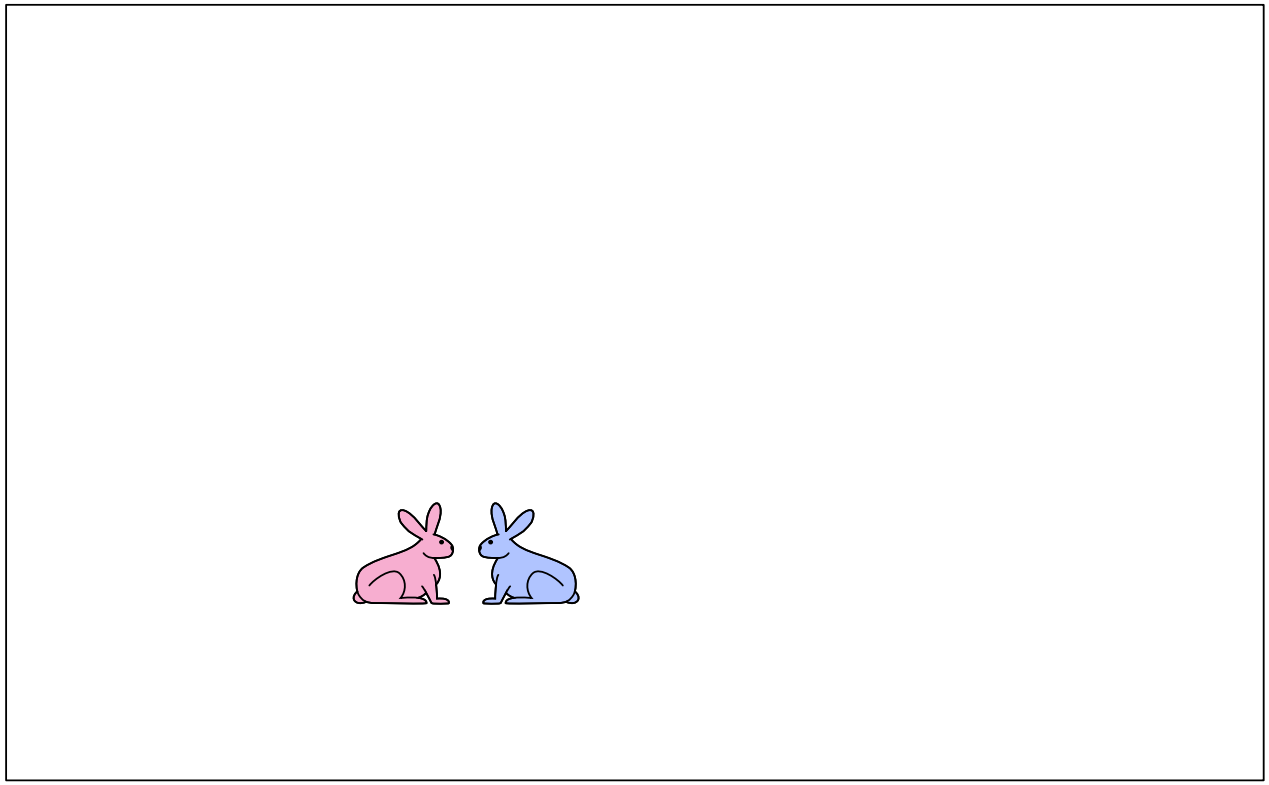
- Un par de conejos recién nacidos (una hembra y un macho) se ponen en un corral
- Los conejos se aparean a la edad de un mes
- Los conejos tienen un período de gestación de un mes
- Suponga que los conejos nunca mueren, que la hembra siempre produce un nuevo par (un macho, una hembra) cada mes desde su segundo mes en adelante.
- ¿Cuántas conejas hay al final de un año?

Fibonacci



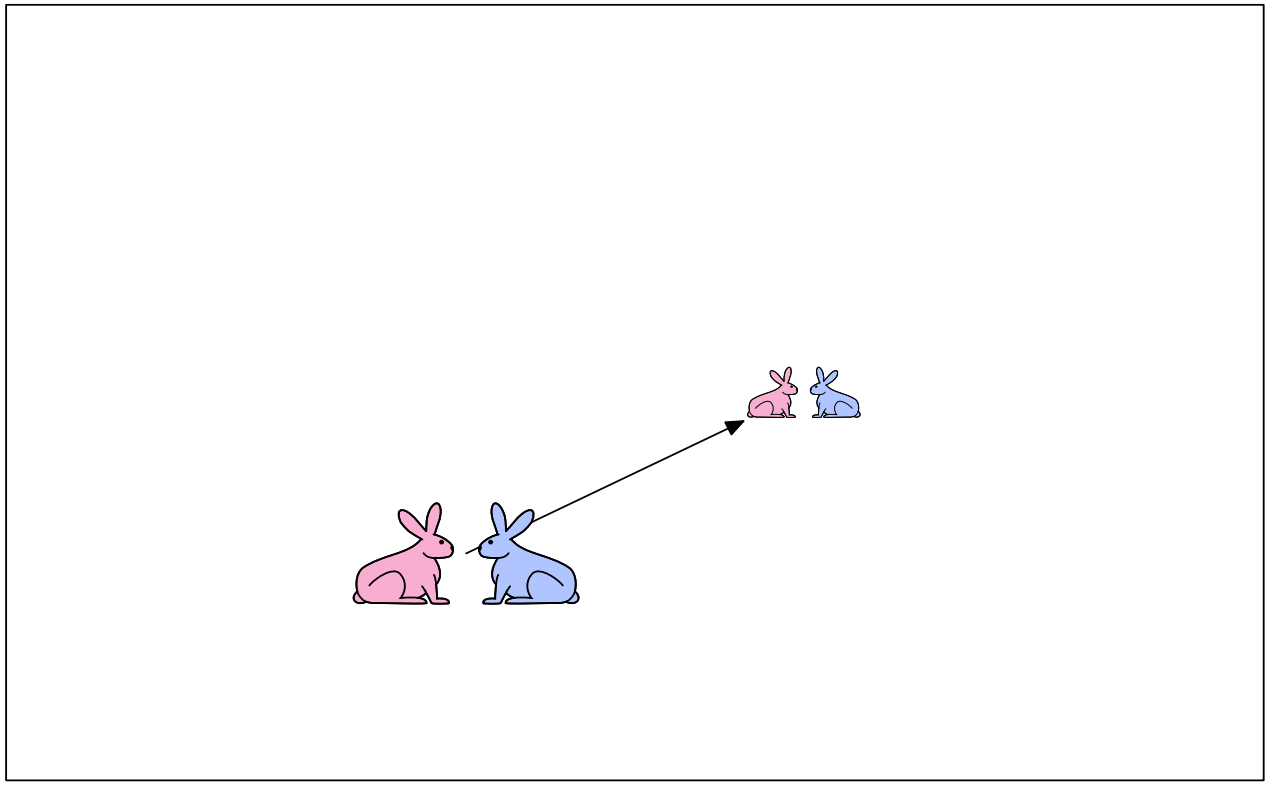
Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci



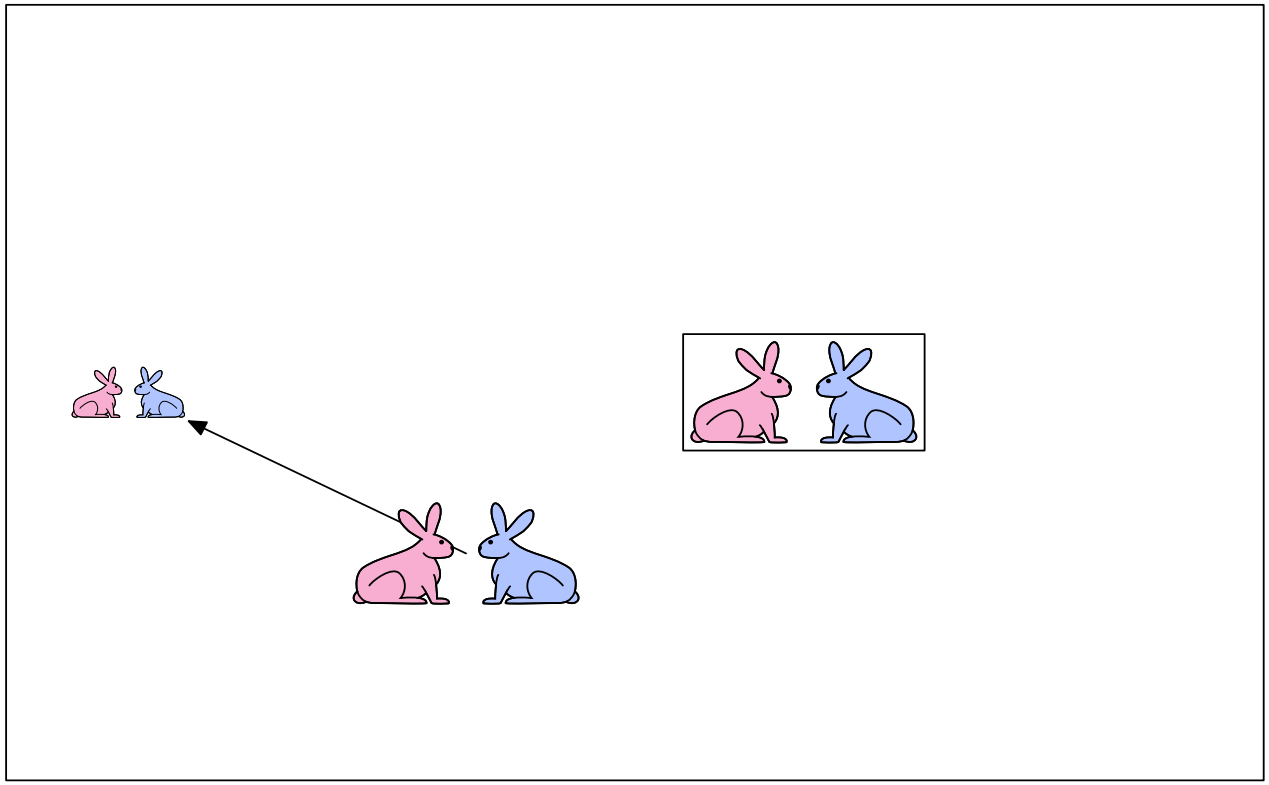
Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci



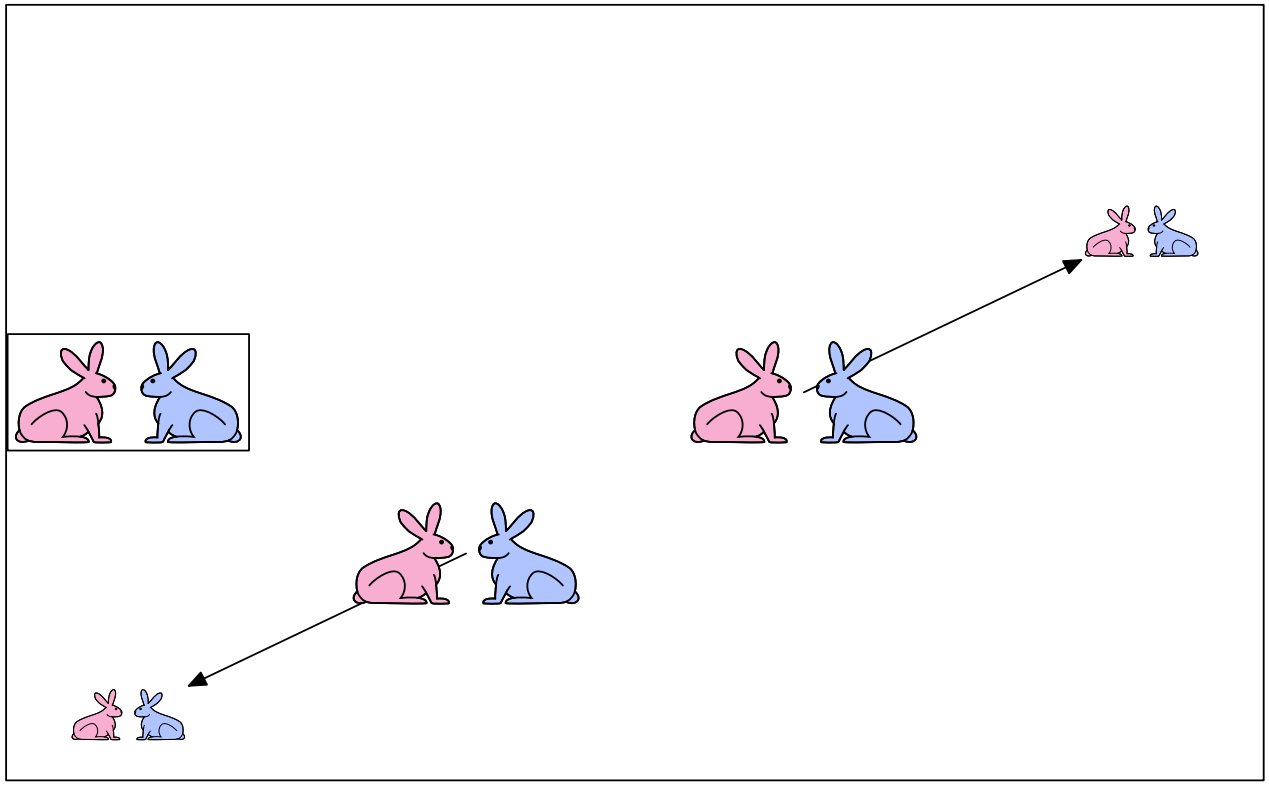
Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci



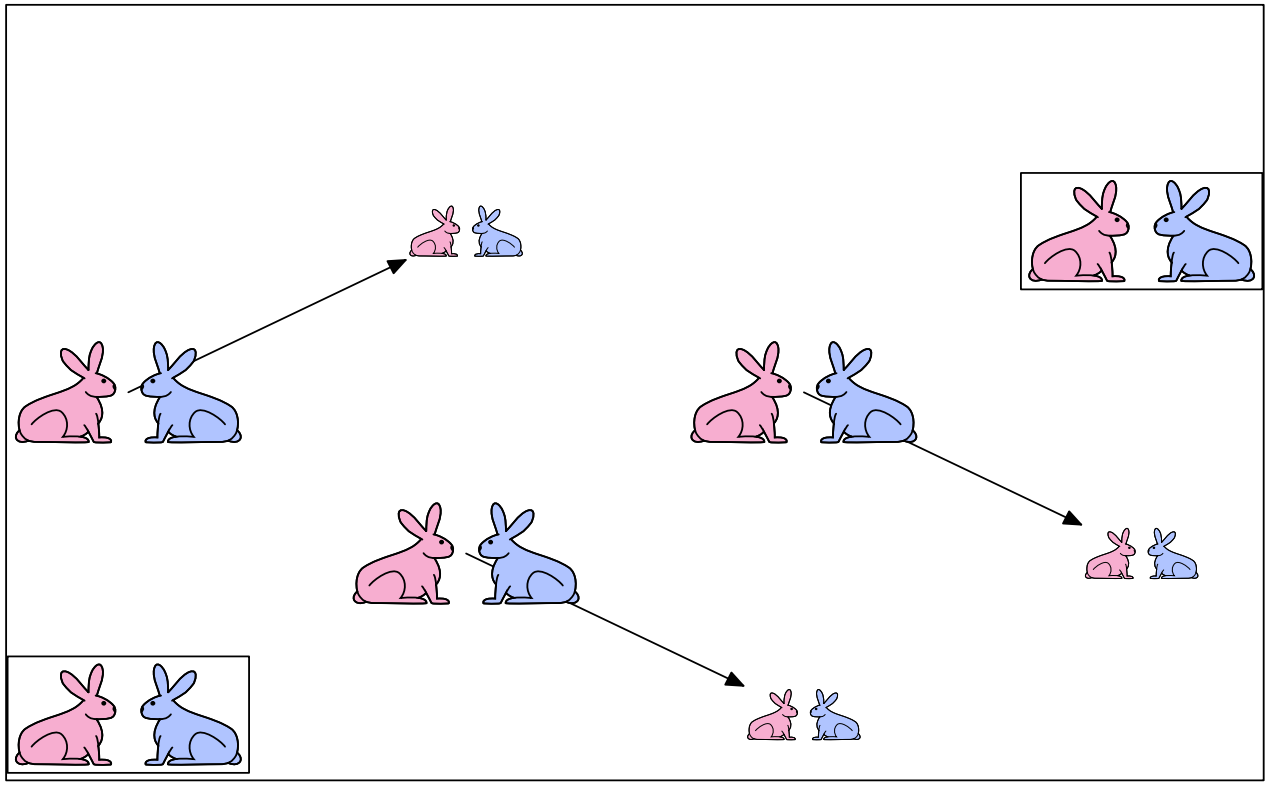
Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci



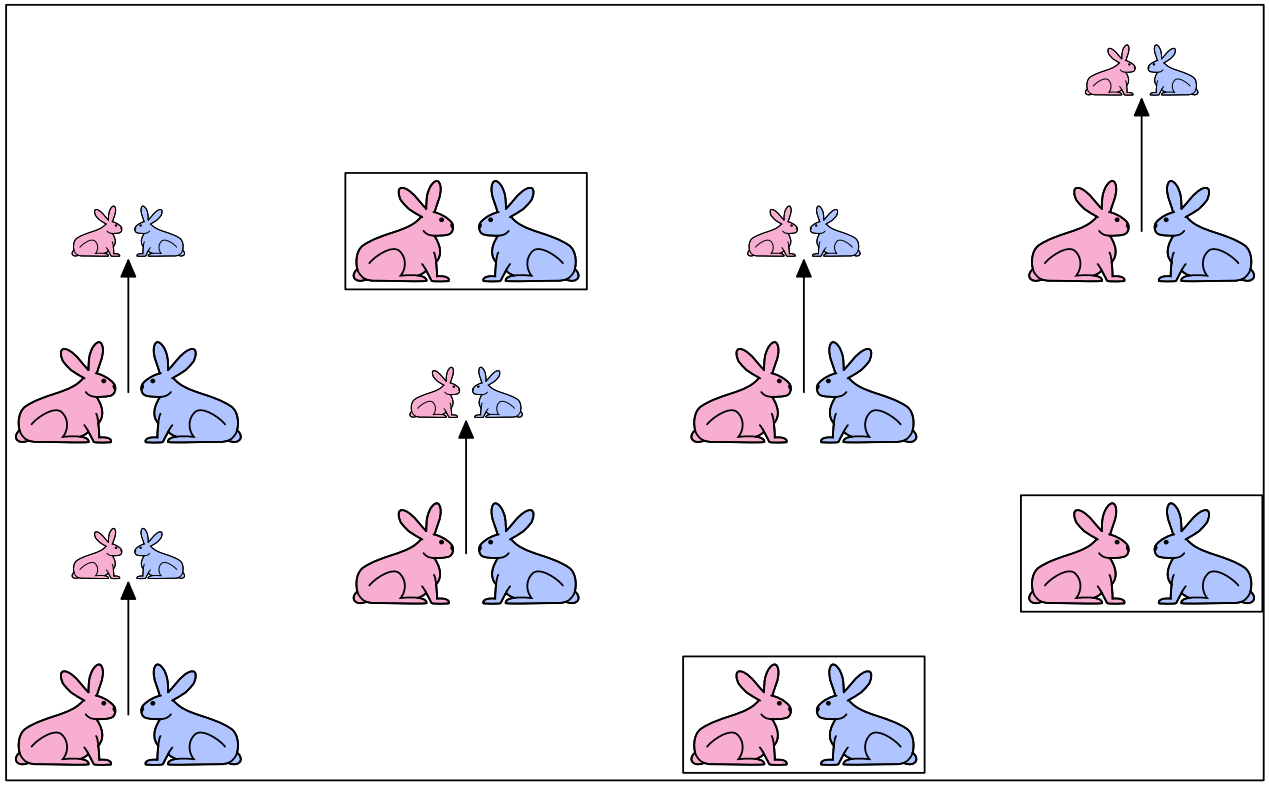
Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci



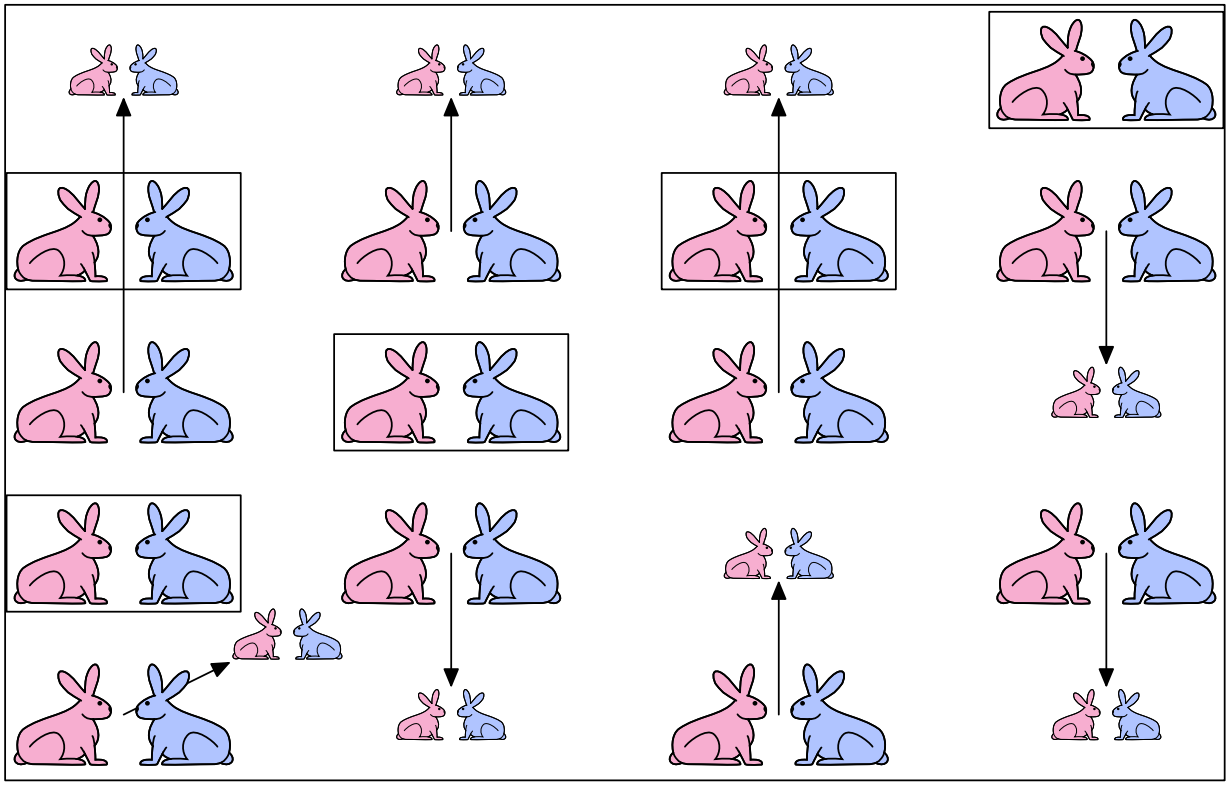
Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci

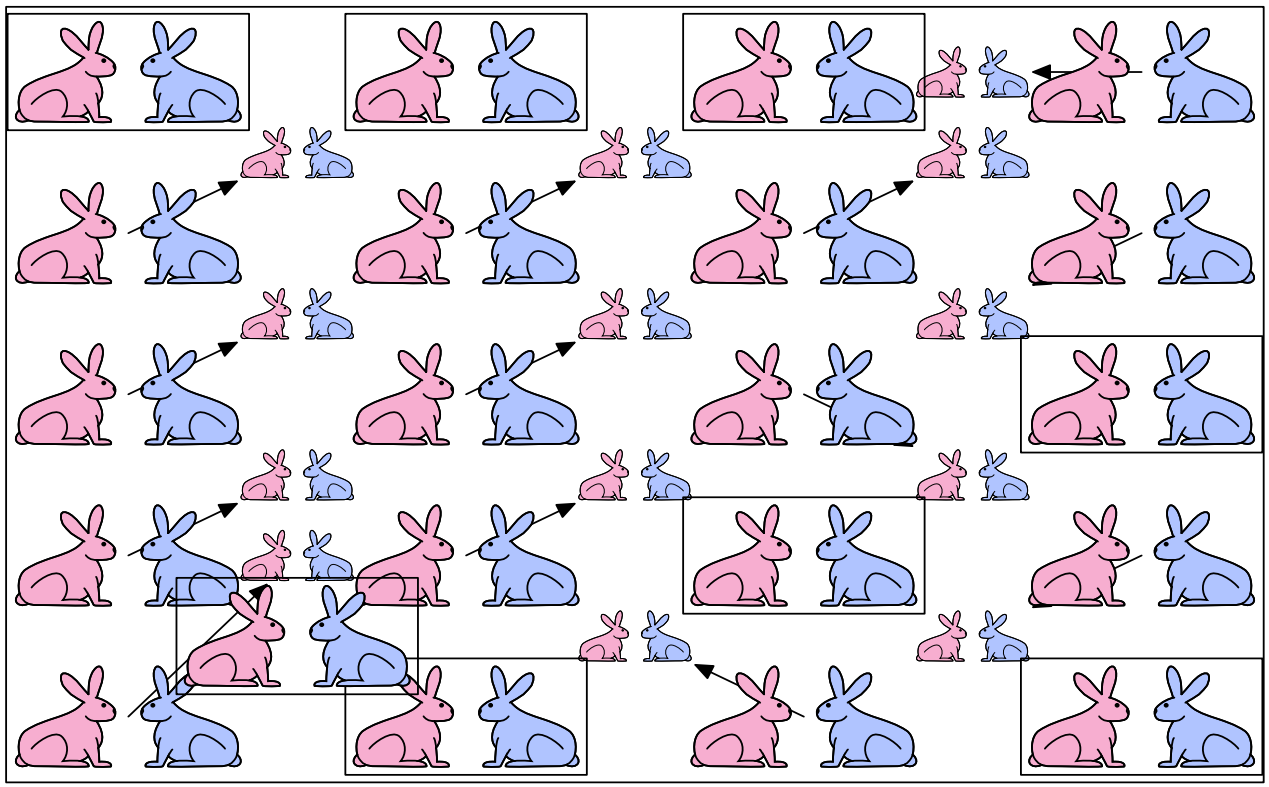


Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci

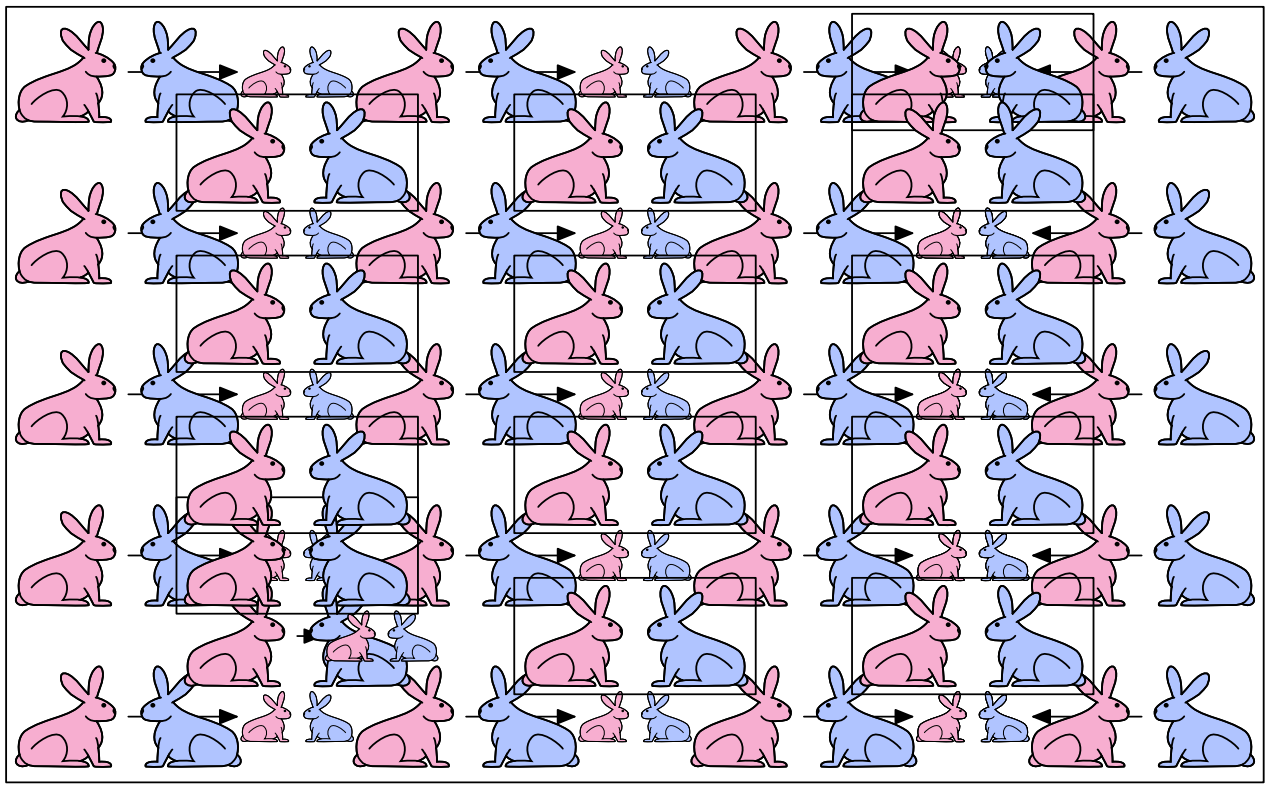


Fibonacci



Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci



Demo courtesy of Prof. Denny Freeman and Adam Hartz

Fibonacci

- Sucesión de eventos:
  - Luego de un mes — 1 hembra.
  - Luego de un segundo mes — 1 hembra (ahora preñada).
  - Al tercer mes - 2 hembras, una preñada y otra no.
- En general, $f(n) = f(n-1) + f(n-2)$:
  - Cada hembra viva en $n-2$ produce una hembra en el mes n.
  - Estas se suman a las hembras en el mes $n-1$.

Fibonacci

- **Casos base**: $f(0) = 1$ y $f(1) = 1$.
- **Caso recursivo**: $f(n) = f(n-1) + f(n-2)$.

```
def fib(x):
    """assumes x an int >= 0
       returns Fibonacci of x"""
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)
```

- Ver en Python Tutor

Palíndromos

- ¿Cómo revisamos si un *string* es un palíndromo (se lee igual al derecho y al revés).
- Primero, obtenemos los caracteres:
  - Quitamos puntuación.
  - Ajustamos mayúsculas y minúsculas
- Luego:
  - **Caso base:** *string* de largo 0 o 1 es palíndromo.
  - **Caso recursivo:** si primer y último caracter coinciden, es palíndromo si la sección del centro es palíndromo.

Palíndromo

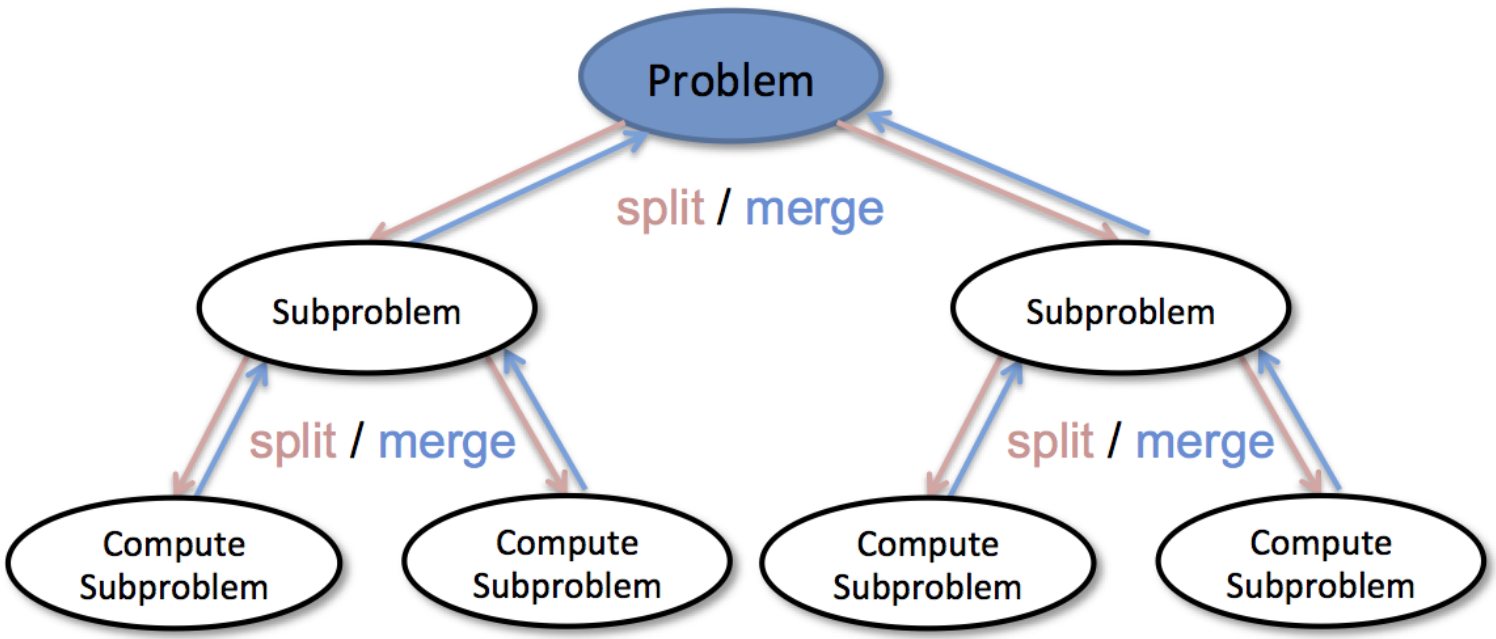
- ‘Able was I, ere I saw Elba’ → ‘ablewasiereisawleba’
- `isPalindrome('ablewasiereisawleba')` is same as
  - `'a' == 'a'` and `isPalindrome('blewasiereisawleb')`

Palíndromo

```
def isPalindrome(s):

    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])

    return isPal(toChars(s))
```

Divide y conquistarás

- Este es también un ejemplo de la estrategia **divide y conquistarás**.
- Podemos resolver un problema difícil descomponiéndolo en partes tales que:
  - Los **subproblemas son más fáciles de resolver**.
  - Las soluciones a los subproblemas se pueden **combinar** para resolver el problema original.



Ejercicio

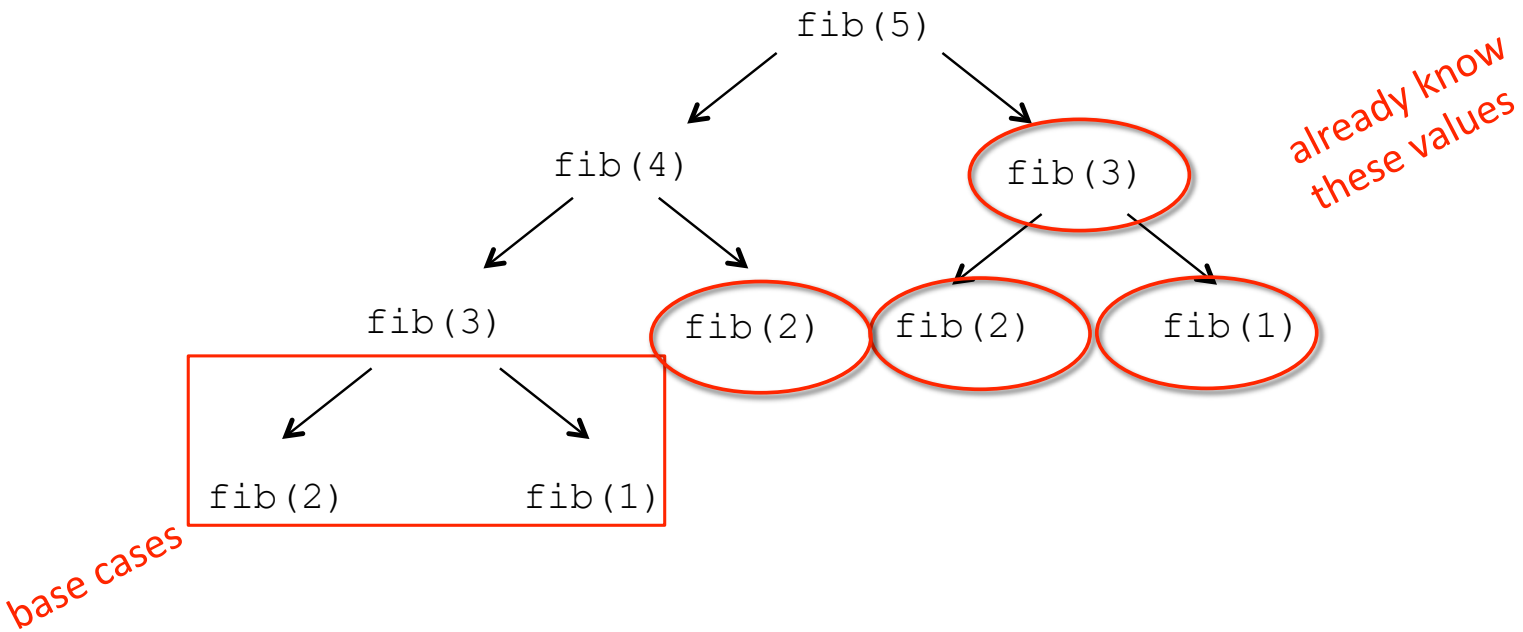
- Escribir una función recursiva que implemente el algoritmo de Euclides para encontrar el *máximo común divisor* (mcd) de dos números a y b.
- ¿Cuál es el caso base?
- ¿Cuál es el paso recursivo?
- Tip: busque el vídeo de *Derivando* en Youtube para guiarse con la solución.

Eficiencia: Fibonacci con diccionarios

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

- two base cases
- calls itself twice
- this code is inefficient

Eficiencia: Fibonacci con diccionarios



- **recalculating** the same values many times!
- could keep **track** of already calculated values

Eficiencia: Fibonacci con diccionarios

```
def fib_efficient(n, d):
    if n in d:
        return d[n]
    else:
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
        return ans

d = {1:1, 2:2}
print(fib_efficient(6, d))
```

Method sometimes called " memoization"

Initialize dictionary with base cases

- do a **lookup first** in case already calculated the value
- **modify dictionary** as progress through function calls

Eficiencia: Fibonacci con diccionarios

- Calling fib(34) results in 11,405,773 recursive calls to the procedure
- Calling fib_efficient(34) results in 65 recursive calls to the procedure
- Using dictionaries to capture intermediate results can be very efficient
- But note that this only works for procedures without side effects (i.e., the procedure will always produce the same result for a specific argument independent of any other computations between calls)