PRÁCTICA 1 EN PYTHON

10/10/2022

DIMITRIS (ERASMUS)

EDUARDO JUNOY ORTEGA

GRUPO 14
LABORATORIO 02

I-A. Multiplicación de matrices

1. Definición de la función matrix_multiplication:

```
def matrix_multiplication(m_1: np.ndarray, m_2: np.ndarray)-> np.ndarray:
    n_rows, n_interm, n_columns = \
        m_1.shape[0], m_2.shape[0], m_2.shape[1]
    m_product = np.zeros( (n_rows, n_columns) )
    for p in range(n_rows):
        for q in range(n_columns):
            for r in range(n_interm):
                m_product[p, q] += m_1[p, r] * m_2[r, q]
    return m_product
```

2. Medición de tiempos con matrix_multiplication, mostrándolos por pantalla:

```
l_timings = []
for i in range(11):
    dim = 10+i
    m = np.random.uniform(0., 1., (dim, dim))
    timings = %timeit -o -n 10 -r 5 -q matrix_multiplication(m, m)
l_timings.append([dim, timings.best])
print(l_timings)
```

I-B. Búsqueda binaria

1. Definición de la función recursiva de busqueda binaria:

```
def rec_bb(t: List, f: int, l: int, key: int)-> int:
    if l >= f:
        m = (f + l) // 2
        if t[m] == key:
            return m
        elif t[m] > key:
            return rec_bb(t, f, m -1, key)
        else:
            return rec_bb(t, m+1, l, key)
    else:
        return None
```

2. Definición de la función iterativa de búsqueda binaria:

```
def bb(t: List, f: int, l: int, key: int) -> int:
    while f <= l:
        mid = (f + l) // 2
        if key == t[mid]:
            return mid
        elif key < t[mid]:
            l = mid-1
        else:
            f = mid+1
```

3. <u>Medición de tiempos en las funciones anteriores:</u>

bb:

```
l_times = []
for i, size in enumerate(range(5, 15)):
    t = list(range(2**i * size))
    key = t[-1]
    timings = %timeit -n 100 -r 10 -o -q bb(t, 0, len(t) - 1, key)
    l_times.append([len(t), timings.best])
times = np.array(l_times)
print(times)
```

bb_rec:

```
l_times = []
for i, size in enumerate(range(5, 15)):
    t = list(range(2**i * size))
    key = t[-1]
    timings = %timeit -n 100 -r 10 -o -q rec_bb(t, 0, len(t) - 1, key)
    l_times.append([len(t), timings.best])
times = np.array(l_times)
print(times)
```

El tiempo es máximo con key = t[-1] porque obliga al árbol a llegar hasta el último elemento del array.

I-C. Cuestiones

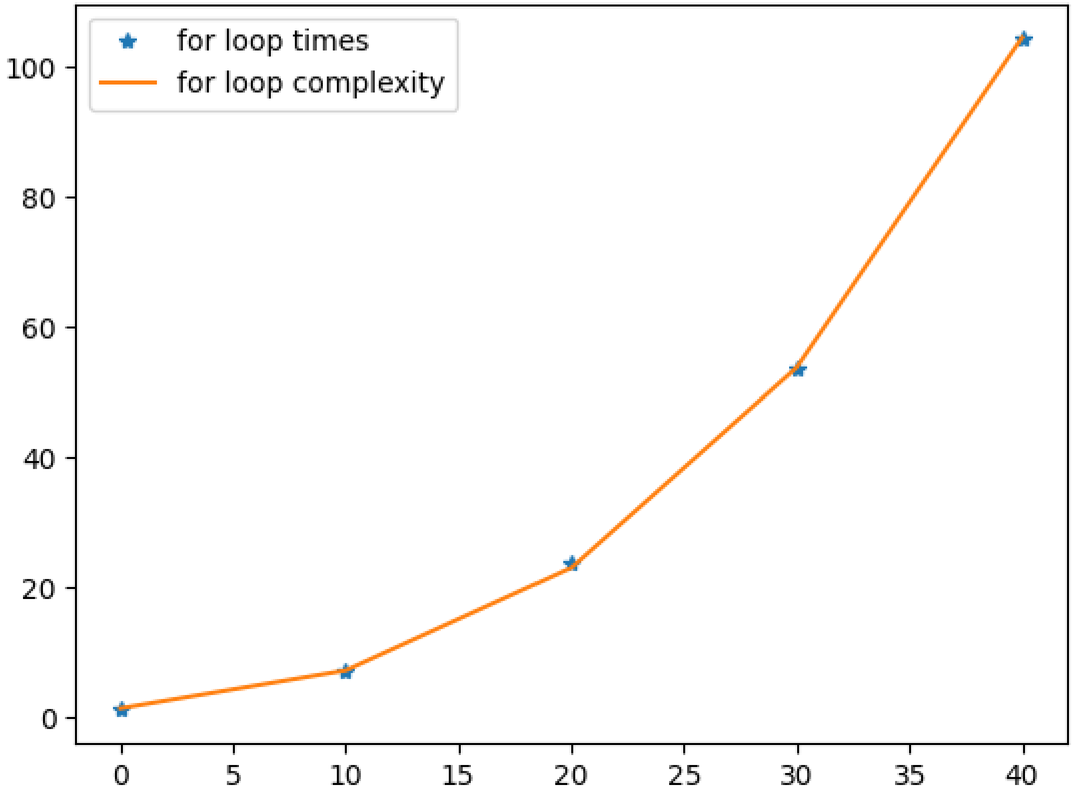
1. <u>La función que se ajusta mejor es n al cubo.</u>

```
def fit_func_2_times(timings: np.ndarray, func_2_fit: Callable):
    if len(timings.shape) == 1:
        timings = timings.reshape(-1, 1)
    values = func_2_fit(timings[ :, 0]).reshape(-1, 1)
    #normalizar timings
    times = timings[ : , 1] / timings[0, 1]
    #ajustar a los valores en times un modelo lineal sobre los valores en values
    lr_m = LinearRegression()
    lr_m.fit(values, times)
    return lr_m.predict(values)

def func_2_fit(n):
    return  n**3
```

El resultado de aplicar la interfaz gráfica a cada función es este:

matrix_multiplication:

```
ran= range(0, 50, 10)
l_timings = []

for i in ran:
    dim = 10+i
    m = np.random.uniform(0., 1., (dim, dim))
    timings = %timeit -o -n 10 -r 5 -q matrix_multiplication(m, m)
    l_timings.append([dim, timings.best])

plt.plot(ran, [elem[1]/l_timings[0][1] for elem in l_timings], '*', label = "loop times")
plt.plot(ran, fit_func_2_times(np.array(l_timings), func_2_fit), label = "loop complexity")
plt.legend()
plt.show()
```

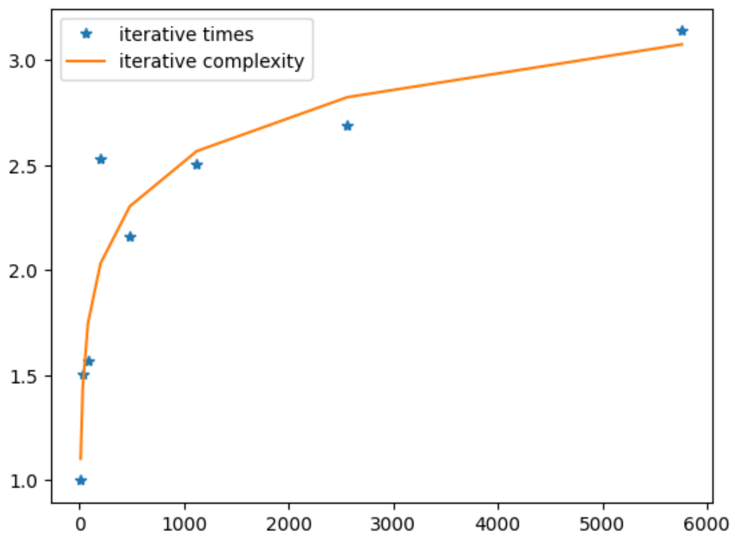


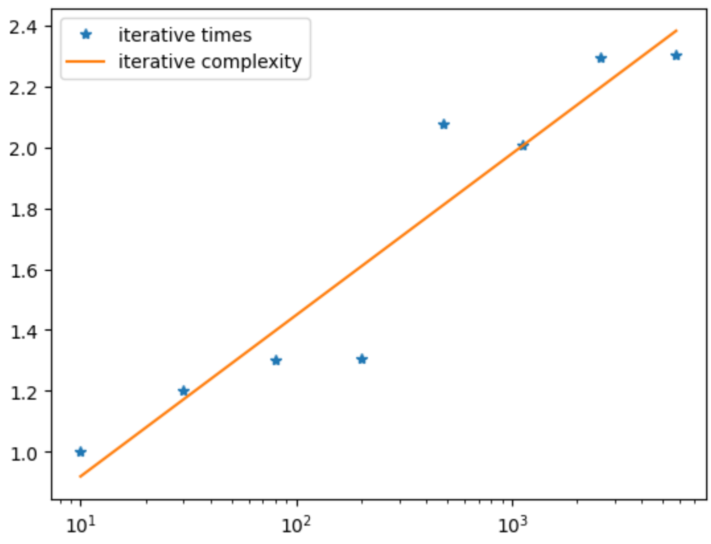
Se puede observar que el tiempo invertido crece exponencialmente con el tamaño de la matriz.

binary_search:

```
def log_fit_function(n):
  return np.array(list(map(math.log2, n)))
l_times = []

for i, size in enumerate(range(10, 50, 5)):
    t = list(range(2**i * size))
    key = t[-1]
    timings = %timeit -n 100 -r 10 -o -q rec_bb(t, 0, len(t) - 1, key)
    l_times.append([len(t), timings.best])
times = np.array(l_times)

x=times[:,0]

plt.plot(x,times[:,1]/times[0][1], '*', label = "iterative times")
plt.plot(x, fit_func_2_times(times, log_fit_function), label = "iterative complexity")
plt.xscale('log')
plt.legend()
plt.show()
```

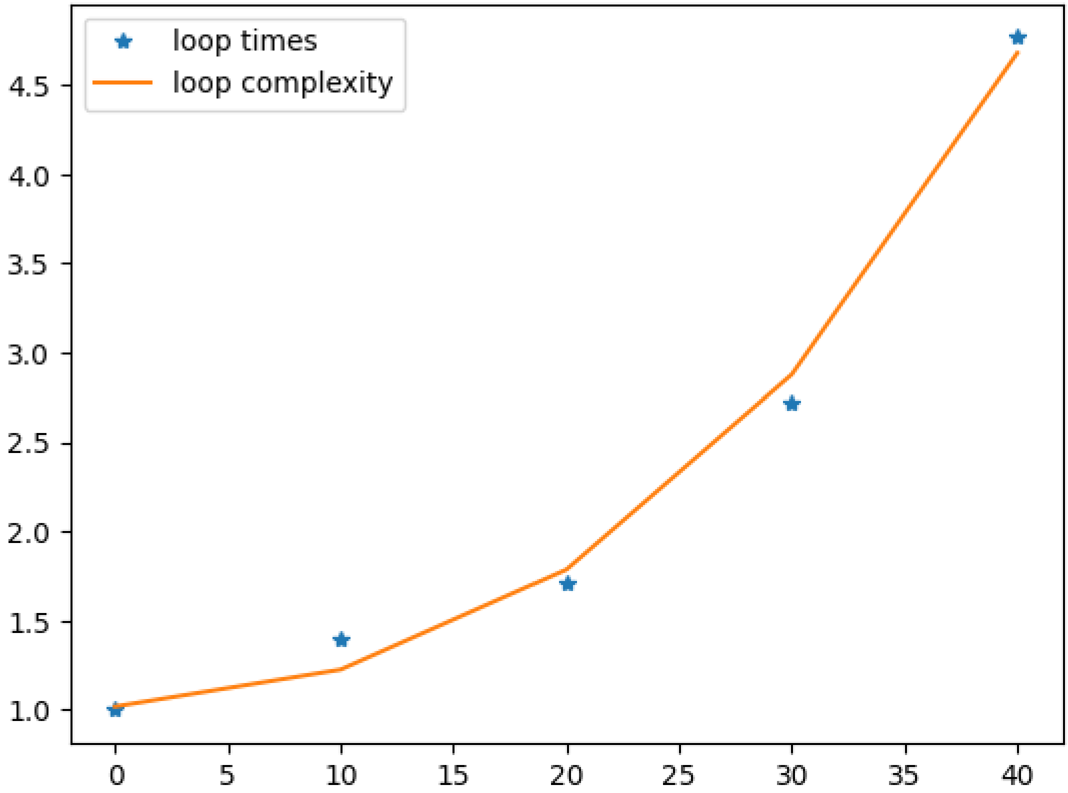
Un árbol, sin embargo, tarda exponencialmente menos en aumentar su tiempo (log2).
Mostrado a escala natural:
Mostrado a escala logarítmica (se asemeja a una función tipo n):





2. <u>Utilizando a.dot para la multiplicación de matrices:</u>

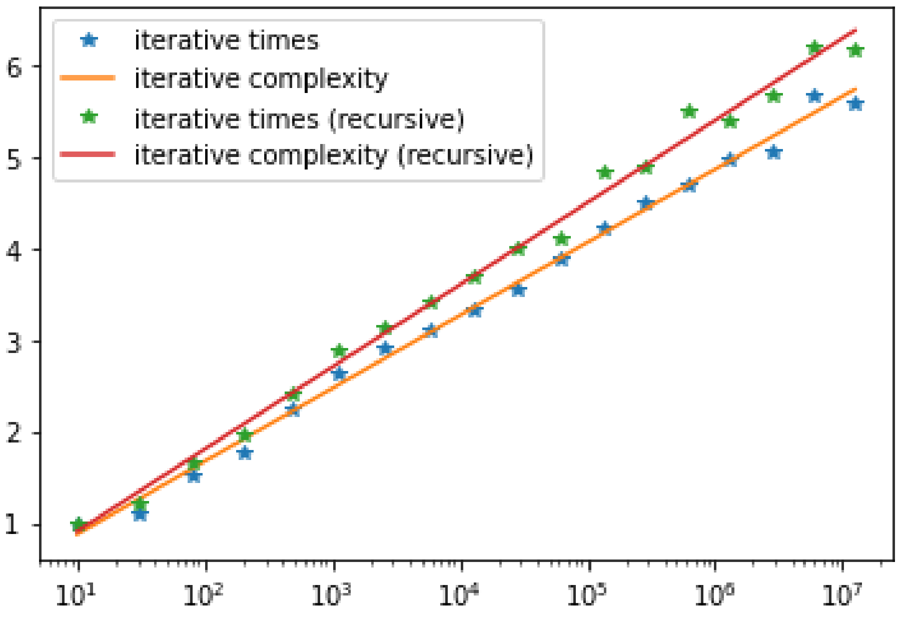
```
ran= range(0, 50, 10)
l_timings = []

for i in ran:
    dim = 10+i
    m = np.random.uniform(0., 1., (dim, dim))
    timings = %timeit -o -n 10 -r 5 -q m.dot(m)
    l_timings.append([dim, timings.best])


plt.plot(ran, [elem[1]/l_timings[0][1] for elem in l_timings], '*', label = "loop times")
plt.plot(ran, fit_func_2_times(np.array(l_timings), func_2_fit), label = "loop complexity")
plt.legend()
plt.show()
```



3. <u>Comparativa entre la búsqueda binaria iterativa y la recursiva:</u>

```
def log_fit_function(n):
  return np.array(list(map(math.log2, n)))

l_times_rec = []
l_times = []

for i, size in enumerate(range(10, 100, 5)):
    t = list(range(2**i * size))
    key = t[-1]
    timings = %timeit -n 20 -r 10 -o -q bb(t, 0, len(t) - 1, key)
    l_times.append([len(t), timings.best])
    timings2 = %timeit -n 20 -r 10 -o -q rec_bb(t, 0, len(t) - 1, key)
    l_times_rec.append([len(t), timings2.best])

times = np.array(l_times)
times_rec = np.array(l_times_rec)

x=times[:,0]

plt.plot(x,times[:,1]/times[0][1], '*', label = "iterative times")
plt.plot(x, fit_func_2_times(times, log_fit_function), label = "iterative complexity")
plt.plot(x,times_rec[:,1]/times_rec[0][1], '*', label = "iterative times (recursive)")
plt.plot(x, fit_func_2_times(times_rec, log_fit_function), label = "iterative complexity
(recursive)")
plt.xscale('log')
plt.legend()
plt.show()
```

Se puede observar cómo a largo plazo la complejidad iterativa es mucho más eficiente.



II-A Min Heaps sobre arrays.

1. Definir min_heapify:

```
def heapify(h: List, i: int):
    while 2*i+1 < len(h):
        n_i = i
        if h[n_i] > h[2*i+1]:
            n_i = 2*i+1
        if 2*i+2 < len(h) and h[n_i] > h[2*i+2]:
            n_i = 2*i+2
        if n_i > i:
            h[i], h[n_i] = h[n_i], h[i]
        else:
            i = n_i
        return h

def min_heapify(h: np.ndarray, i: int):
    heapify(h, i)
```

2. Definir insert_min_heap:

```
def insert_min_heap(h: np.ndarray, k: int)-> np.ndarray:
    if h == None:
        h = []
    h += [k]
    j = len(h) - 1
    while j >= 1 and h[(j-1) // 2] > h[j]:
        h[(j-1) // 2], h[j] = h[j], h[(j-1) // 2]
        j = (j-1) // 2
```

3. Definir create_min_heap:

```
def create_min_heap(h: np.ndarray):
    j = h[(j-1) // 2](len(h) - 1)
    while j > -1:
        min_heapify(h, j)
        j -=1
    return h
```

II-B. Colas de prioridad sobre Min Heaps.

1. Definir la función pq_ini():

```
def pq_ini():
    pq = []
    return pq
```

2. Definir la función pq_insert:

```
def pq_insert(h: np.ndarray, k: int)-> np.ndarray:
    h += [k]
    j = len(h) -1
    while j >= 1 and h[(j-1) // 2] > h[j]:
        h[(j-1) // 2], h[j] = h[j], h[(j-1) // 2]
        j = h[(j-1) // 2]
    return h
```

3. Definir la función pq_remove:

```
def pq_remove(h: np.ndarray)-> Tuple[int, np.ndarray]:
    if len(h) == 0:
        return (h[0], h)
    e = h[0]
    h[0] = h[-1]
    h.pop()
    min_heapify(h, 0)
    return (e, h)
```

II-C. El problema de selección.

```
def select_min_heap(h: np.ndarray, k: int)-> int:
    h = [-i for i in h]
    l = h[k:]
    h = h[:k]
    for i in l:
        if h[0] < 1:
            h[0] = i
            min_heapify(h, 0)
    return -h[0]
```

II-D. Cuestiones

1. Calcular tiempos de la función min_heaps:

```
l_times = []

for i, size in enumerate(range(10, 50, 5)):
    t = list(range(2**i * size))
    key = t[-1]
    timings = %timeit -n 100 -r 10 -o -q min_heapify(t, 0, len(t) - 1, key)
    l_times.append([len(t), timings.best])
times = np.array(l_times)

x=times[:,0]

plt.plot(x,times[:,1]/times[0][1], '*', label = "iterative times")
plt.plot(x, fit_func_2_times(times, log_fit_function), label = "iterative complexity")
plt.legend()
plt.show()
```

Se asemeja a una función tipo n.

2. Hallar el coste de la función:

$O(k + \ (n - k) log\ k)$

3. Ventaja del problema de selección:

La ventaja se da por la propia naturaleza de la función del algoritmo, que supone un menor número de combinaciones.

Los k elementos previamente insertados se obtienen en orden, de tal manera que el primer elemento para extraer el primer elemento sería k, el segundo k-1, el tercero k-2 y así continuamente hasta llegar al último elemento.

4. Obtener los dos menores elementos de un array es mediante un for:

```
f_min = s_min = lista[0]
for e in lista:
        if e<f_min:
                s_min = f_min
                f_min = e
…
```