PRÁCTICA 1: Python Básico. Min heaps

Algoritmos y Estructura de Datos Avanzada 2022-2023

Pareja 11:

Jose Manuel García Giráldez

Alejandro Monterrubio Navarro

I. Python Básico

a. Midiendo tiempos con %timeit

En este apartado se nos pide usar %timeit para medir tiempos de ejecución en funciones, para nuestro caso usamos multiplicación de matrices, el código desarrollado es el siguiente:

Multiplicación de matrices:

```
def matrix_multiplication(m_1: np.ndarray, m_2: np.ndarray) ->
np.ndarray :

   result = np.zeros((len(m_1),len(m_1)))

   for i in range(len(m_1)):

      for j in range(len(m_2[0])):

         for k in range(len(m_2)):
            result[i][j] += m_1[i][k] * m_2[k][j]

   return result
```

Luego usando el siguiente código medimos los tiempos:

```
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)
```

Los resultados son:

[[10, 0.0007895599999997671], [11, 0.0010678600000005646], [12, 0.0013637000000002787], [13, 0.0017135100000004399], [14, 0.002146659999999656], [15, 0.0026473199999998086], [16, 0.0032358700000003184], [17, 0.0038597600000002787], [18, 0.004538610000000176], [19, 0.005362620000001072], [20, 0.006229690000000687]]

b. Búsqueda binaria

En este apartado se pide programar una búsqueda binaria recursiva y otra iterativa, los códigos de estas son:

Búsqueda Binaria Recursiva:

```
def rec_bb(t: list, f: int, l: int, key: int) -> int :

   if l>=f:

      mid = (f+l) // 2

      if t[mid] == key:
         return mid

      elif t[mid] < key:
         return rec_bb(t, mid+1, l, key)

      else:
         return rec_bb(t, f, mid-1, key)

   else:
      return None
```

Búsqueda Binaria Iterativa

```
def bb(t: list, f: int, l: int, key: int) -> int :

   while f <= l:

      mid = (f+l) // 2

      if t[mid] == key:
         return mid

      elif t[mid] < key:
         f = mid + 1

      elif t[mid] > key:
         l = mid -1

   return None
```

Para medir los tiempos usamos el siguiente código, se cambia la X por rec_bb o bb según lo que necesitemos:

```python
l_times = []
for i, size in enumerate(range(5, 15)):
   t = list(range(2**i * size))
   key = 8
   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(l_times)
```

Tiempos para rec_bb:

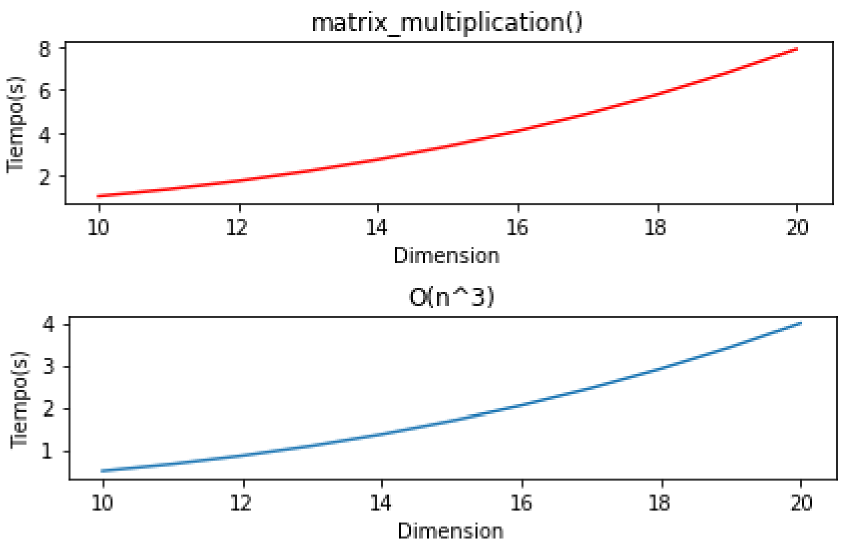
[[5, 7.679999998799758e-07], [12, 4.939999996622646e-07], [28, 1.0140000006231276e-06], [64, 1.1610000001383014e-06], [144, 7.900000002791785e-07], [320, 1.694000000043161e-06], [704, 1.7519999994419778e-06], [1536, 1.8889999989823992e-06], [3328, 1.998999999841544e-06], [7168, 2.5419999997211563e-06]]

Tiempos para bb:

[[5, 5.490000000918372e-07], [12, 4.1999999893960194e-07], [28, 8.359999992535449e-07], [64, 1.0029999998550921e-06], [144, 7.160000006933842e-07], [320, 1.5039999993859965e-06], [704, 1.4790000000175496e-06], [1536, 1.5680000001339067e-06], [3328, 1.8799999998009299e-06], [7168, 2.2249999994983226e-06]]

c. Cuestiones

Muestra de las gráficas para su comparación (código en P111Scripts.py):



II. Min Heaps

a. Min Heaps sobre arrays de Numpy

1. Nos piden crear una función que haga heapify a un elemento i en un array h, el código es:

Min heap:

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

   #Mientras no estes en el nodo hoja
   while 2*i+1 < len(h):
      n_i = i

      #Si nodo enviado > que hijo izquierdo entonces guardas posicion
hijo
      if h[i] > h[2*i+1]:
         n_i = 2*i+1

      #Comprueba si hay hijo derecho y si lo hay comprueba nodo
estamos > derecho y si posicion guardada es > hijo derecho, si ambos
son mayores se actualiza posicion
      if 2*i+2 < len(h) and h[i] > h[2*i+2] and h[2*i+2] < h[n_i]:
         n_i = 2*i+2

      #Si no hijo izq y drc es menor, y nodo que hemos guardado tiene
valor menor o posicion baja, intercambia valores de i y n_i
      if n_i > i:
         h[i], h[n_i] = h[n_i], h[i]
         i = n_i

      #Si no hace nada, devuelve
      else:
         return
```

2. Ahora se pide crear una función que inserte un entero k en el min h y devuelva el nuevo min heap, el código es:

Insert Min Heap:

```
def insert_min_heap(h: np.ndarray, k: int) -> np.ndarray:
   if h is None:
      return [k]
   #primero añade el elemento al heap
   h = np.append(h,k)
   j = len(h) - 1

   # coloca el elemento en su lugar correspondiente
   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

    return h
```

3. Por último, se pide crear una función que cree un min heap sobre un array numpy pasado como argumento, el código es:

Create Min Heap:

```
def create_min_heap(h: np.ndarray):

   # opcion 1: realizar un heapify sobre el array pasado por
argumento tantas veces como sea necesario
   # esto permite evitar la necesidad de realizar un segundo array,
de arriba a abajo
   """j = (len(h)-1) // 2
   for i in range(0,j+1):
      for k in range(0,i+1):
         min_heapify(h,k)

   return h"""

   # opcion 2: realiza un heapify de los padres de todos los
subarboles de abajo arriba
   k = (len(h)-1)//2
   while k >= 0:
      min_heapify(h, k)
      k-=1
   return h
```

b. Colas de prioridad sobre Min Heaps

Ahora vamos a usar las funciones anteriores sobre min heaps para crear primitivas de unas colas de prioridad y donde el valor de cada elemento coincide con su prioridad.

1. Primero creamos una función que inicialice una cola de prioridad vacía. El código es:

Priority Queue Init:

```
def pq_ini() -> np.ndarray:
   q = []
   return q
```

2. Creamos una función que inserte el elemento k en la cola de prioridad h y devuelva la nueva cola. El código es:

Priority Queue Insert:

```
def pq_insert(h: np.ndarray, k: int)-> np.ndarray:
```

```
    return insert_min_heap(h, k)
```

3. Ahora creamos una función que elimina el elemento con menor prioridad y devuelve ese elemento y la nueva cola. El código es:

Priority Queue Remove:

```
def pq_remove(h: np.ndarray)-> np.ndarray:
    if h is None or len(h) == 0: return None

    elim = h[0]
    h = np.delete(h,0)
    min_heapify(h,0)
    return elim, h
```

c. El problema de selección

Para solucionar el problema de la selección y crear la función "select_min_heap" hemos desarrollado el siguiente código:

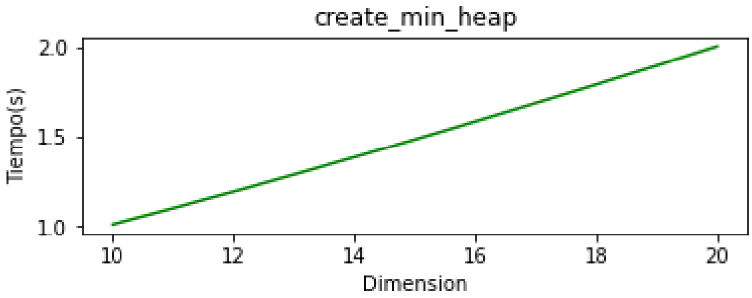
Select min Heap:

```
def select_min_heap(h: np.ndarray, k: int)-> int:

    aux = h.copy()
    # invierte el array
    aux = np.multiply(aux, -1)
    # se cogen los k primeros
    aux_mh = aux[:k]

    # realiza el min_heap sobre el array invertido
    aux_mh = create_min_heap(aux_mh)
    for i in range (k, len(h)):
        if aux[i] > aux_mh[0]:
            aux_mh[0] = aux[i]
            min_heapify(aux_mh, 0)

    return aux_mh[0]*-1
```

d. Cuestiones

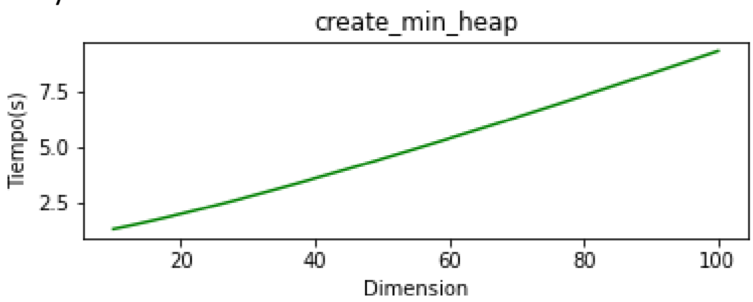
1. Analizar visualmente los tiempos de ejecución de nuestra función de creación de min heaps. ¿A qué función f se deberían ajustar dichos tiempos?

Con un script obtenemos la siguiente función producto de utilizar dimensiones entre 10 y 20 para llamar a create_min_heap.

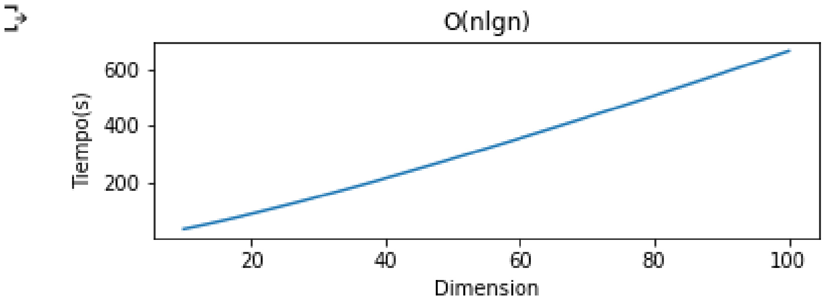


Sin embargo, aquí no se nota mucho de que función podría tratarse así que aumentamos el rango a una dimensión de 100.

Entre 10 y 100:



Aquí podemos ver una ligera curvatura. Si representamos la función de nlogn veremos que es muy similar:



2. Expresar en función de k y del tamaño del array cual debería ser el coste de nuestra función para el problema de selección.

Nos fijamos en el código:

```
def select_min_heap(h: np.ndarray, k: int)-> int:

    aux = h.copy()
    # invierte el array
    aux = np.multiply(aux, -1)
    # se cogen los k primeros
    aux_mh = aux[:k]

    # realiza el min_heap sobre el array invertido
    aux_mh = create_min_heap(aux_mh)
```

```
    for i in range (k, len(h)):
        if aux[i] > aux_mh[0]:
            aux_mh[0] = aux[i]
            min_heapify(aux_mh, 0)

    return aux_mh[0]*-1
```

Partimos de que el coste realizar un heapify es O(logn) y que el coste de llamar a create_min_heap es el coste de realizar heapify (len(h)-1//2) veces, es decir =O(nlogn), como ya vimos en el apartado anterior, con todo esto, viendo que la selección realiza una inversión del array, costando n después una llamada a create_min_heap que cuesta klogk y por último ejecuta en un bucle (len(h) - k) veces min_heapify con coste logk, el coste en función de K y n, siendo n el tamaño del array, sería n + klogk + (n-k)logk, lo que resulta en O(nlogk).

3. Una ventaja de nuestra solución al problema de selección es que también nos da los primeros k elementos de una ordenación del array. Explicar por qué esto es así y como se obtendrían.

Puesto que como podemos observar en nuestro código lo que estamos manteniendo es un heap de los K menores elementos del array cuyos valores se han convertido a su negativo y cuya raíz es el valor que buscamos, es decir, el elemento en la posición K, sabemos que ese heap contiene esos K elementos, menores que el elemento raíz. Para obtenerlos bastaría con devolver el heap al finalizar el bucle. Podemos probar esto con un simple print de la siguiente forma:

```
def select_min_heap(h: np.ndarray, k: int)-> int:

    aux = h.copy()
    # invierte el array
    aux = np.multiply(aux, -1)
    # se cogen los k primeros
    aux_mh = aux[:k]

    # realiza el min_heap sobre el array invertido
    aux_mh = create_min_heap(aux_mh)
    for i in range (k, len(h)):
        if aux[i] > aux_mh[0]:
            aux_mh[0] = aux[i]
            min_heapify(aux_mh, 0)
    print(aux_mh)
    return aux_mh[0]*-1


```

En nuestro caso print(aux_mh), que nos mostrará los K elementos, aunque con valor negativo, haría falta convertirlos de nuevo a positivo. Como podemos ver:

```
______________________________Checking heap based selection
[5, 7, 8, 9, 10, 11, 12, 13, 15, 19]
[19, 8, 10, 11, 9, 7, 13, 15, 5, 12]
[-5]
pos 1   val 5
[19, 8, 10, 11, 9, 7, 13, 15, 5, 12]
[-7, -5]
pos 2   val 7
[19, 8, 10, 11, 9, 7, 13, 15, 5, 12]
[-8, -5, -7]
pos 3   val 8
[19, 8, 10, 11, 9, 7, 13, 15, 5, 12]
[-9, -8, -7, -5]
pos 4   val 9
[2, 5, 7, 9, 13, 14, 15, 16, 17, 19]
[14, 13, 5, 19, 15, 2, 16, 7, 17, 9]
[-2]
pos 1   val 2
```

Por ejemplo, en la posición 3 se encuentra el valor 8, si mostramos aux_mh se mostrarán los valores –8,-5 y –7 que corresponden a los valores de los K elementos primeros mencionados en negativo.

O bien podemos probar el siguiente código:

```python
def select_min_heap(h: np.ndarray, k: int)-> int:

    aux = h.copy()
    # invierte el array
    aux = np.multiply(aux, -1)
    # se cogen los k primeros
    aux_mh = aux[:k]

    # realiza el min_heap sobre el array invertido
    aux_mh = create_min_heap(aux_mh)
    for i in range (k, len(h)):
        if aux[i] > aux_mh[0]:
            aux_mh[0] = aux[i]
            min_heapify(aux_mh, 0)

    return np.multiply(aux_mh, -1)


```

Con el que se obtiene:

```
[2, 5, 6, 9, 10, 12, 13, 14, 17, 19]
pos 1   val [2]
pos 2   val [5 2]
pos 3   val [6 5 2]
pos 4   val [9 5 6 2]
[3, 6, 9, 10, 13, 15, 16, 17, 18, 19]
pos 1   val [3]
pos 2   val [6 3]
pos 3   val [9 6 3]
pos 4   val [10  9  6  3]
[0, 2, 6, 8, 9, 10, 11, 12, 13, 15]
pos 1   val [0]
pos 2   val [2 0]
pos 3   val [6 2 0]
pos 4   val [8 6 0 2]
```

se puede observar el correcto funcionamiento.

4. La forma habitual de obtener los dos menores elementos de un array es mediante un doble for donde primero se encuentra el menor elemento y luego el menor de la tabla restante. ¿Se podrían obtener esos dos elementos con un único for sobre el array? ¿Cómo?

Si, podemos guardar los 2 primeros valores del array y en un for recorrer el resto de los valores comparando de la siguiente forma:

```python
def obtiene_menores(h: np.ndarray)-> np.ndarray:
    m1 = h[0]
    m2 = h[1]
    for i in range(2, len(h)):
        if m1 > h[i] or m2 > h[i]:
            if m1 > m2:
                m1 = h[i]
            else:
                m2 = h[i]

    return [m1,m2]
```

Si lo probamos veremos que funciona:

H = [1,2,3,52,12,43,513,1,0,-1] `[-1, 0]`

H = [4,-3,-4,-5,-12,-2,-1,0,2,-24,-1] `[-12, -24]`

H = [5,4,3,2,1,0,1,2,3,4,5] `[1, 0]`

H = [5,4,3,2,1,0,0,1,2,3,4,5] `[0, 0]`