Python básico. Min Heaps

Algoritmos y Estructuras de Datos Avanzadas 2023-2024

Práctica 1

Fecha de entrega: 20 de octubre de 2023

ÍNDICE

I. PYTHON BÁSICO

I-A. Midiendo tiempos con `%timeit`

Vamos a usar la orden mágica `%timeit` de IPython para medir tiempos de ejecución de pequeñas funciones, en nuestro caso, de multiplicación de matrices.

1. Escribir una función

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

   que reciba dos matrices Numpy de dimensiones compatibles y devuelva otra matriz Numpy con su producto.
2. Usar el pequeño script inferior para medir los tiempos de ejecución de la función anterior sobre matrices aleatorias de dimensión creciente y guardarlos en una lista para poder analizarlos o procesarlos luego:

   ```
   l_timings = []
   for i in range(10, 21):
       dim = 10+i**2
       m = ...
       timings = %timeit -o -n 10 -r 5 -q matrix_multiplication(m, m)
       l_timings.append([dim, timings.best])
   ```

   Para ello cambiar los puntos suspensivos por una llamada adecuada al generador numpy `uniform` de números aleatorios uniformes que devuelva matrices aleatorias de dimensión `dim x dim` con valores aleatorios uniformes entre 0. y 1.

I-B. Búsqueda binaria sobre listas

Vamos a programar y medir tiempos de ejecución de funciones recursivas e iterativas de búsqueda binaria sobre **listas** en Python.

1. Escribir una función

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

   que reciba una lista, sus índices `f, l` primero y último, y una clave `key` , y aplique una versión recursiva de la búsqueda binaria para encontrar la posición de `key` entre `f` y `l` . Si no la encuentra, devolverá `None` .
2. Escribir una versión iterativa

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

   de la función anterior con las mismas especificaciones.
3. Adecuar el script inferior para medir los tiempos de ejecución de las funciones anteriores:

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

a_timings = np.array(l_timings)
```

escogiendo valores adecuados de `key` para que el tiempo de ejecución sea máximo.

I-C. Cuestiones

1. ¿A qué función f se deberían ajustar los tiempos de ejecución de la función de multiplicación de matrices? Suponiendo que el tiempo abstracto de ejecución de un algoritmo es $f(t)$, usar el código inferior para ajustar valores de la forma $a \cdot f(t) + b$ a los tiempos reales de ejecución de dicho algoritmo, tiempos que habremos guardado en el array Numpy `a_timings` , para a continuación dibujar tanto los tiempos como el ajuste calculado por la función, y comentar los resultados.

```
from scipy.optimize import curve_fit

def tofit(x, a, b):
    #cambiar f por el ócdigo Python de la ófuncin de
    return a * f(x) + b

x = a_timings[: , 0]
#o (mas "pythonico"?) x = a_timings.T[0].T
y = a_timings[: , 1]
y = y / y[0]    #normalizar timings

pars, _ = curve_fit(tofit, x, y)
```

2. Calcular los tiempos de ejecución que se obtendrían usando la multiplicación de matrices `a.dot(b)` de Numpy y compararlos con los anteriores.
3. ¿Qué clave resultaría en el caso más costoso de la búsqueda binaria? Comparar los tiempos de ejecución de las versiones recursiva e iterativa de la búsqueda binaria en su caso más costoso y dibujarlos para unos tamaños de tabla adecuados. ¿Qué relación encuentras entre ambos tiempos? Argumentar gráficamente dicha relación.

II. Min Heaps

II-A. Min Heaps sobre arrays de Numpy

En la sección anterior hemos trabajado con listas en Python; ahora vamos a hacerlo con **arrays de Numpy**. Aunque son objetos muy distintos y también lo son sus métodos, el trabajo ordinario con arrays es muy parecido al de listas; en particular, ambos acceden a sus elementos con la notación `a[i]` y la sintaxis y efectos del slicing de índices también es el mismo. Por ejemplo `t[ : : -1]` invierte tanto una lista como un array. Además, es muy probable que el código escrito para la búsqueda binaria sobre listas funcione también sobre arrays.

Sin embargo, y de nuevo, se trata de objetos **muy distintos** y también lo son sus métodos. Por ello, y por ejemplo, si una función que debe devolver una lista devuelve de hecho un array de Numpy, es casi seguro que el código que siga a la llamada y maneje el objeto devuelto acabe en un error. Por tanto, **hay que atenerse estrictamente a las definiciones de funciones que se usen en cada caso**.

En esta sección vamos a implementar sobre arrays de Numpy diversas funciones de trabajo con Min Heaps. Para ello supondremos que un min heap vacío se representa como un array Numpy vacío (por ejemplo, como `np.array([])`).

1. Escribir una función

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

   que reciba un array `h` de Numpy y aplique la operación de heapify al elemento situado en la posición `i` .
2. Escribir una función

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

   que inserte el entero `k` en el min heap contenido en `h` y devuelva el nuevo min heap.
3. Escribir una función

   `create_min_heap(h: np.ndarray)`

   que cree un min heap sobre el array de Numpy pasado como argumento. Esta creación debe hacerse “in-place”, esto es, sobre el propio array sin usar ningún otro auxiliar (y sin devolver nada).

II-B. Colas de prioridad sobre Min Heaps

Vamos a usar las funciones anteriores sobre min heaps para programar las primitivas de colas de prioridad suponiendo que las mismas son arrays de enteros en Numpy y donde el valor de cada elemento coincide con su prioridad. Suponemos también que los elementos con un valor de prioridad menor salen antes que los que tienen prioridad mayor.

1. Escribir una función
   `pq_ini()`
   que inicialice una cola de prioridad vacía.
2. Escribir una función
   `pq_insert(h: np.ndarray, k: int)-> np.ndarray`
   que inserte el elemento `k` en la cola de prioridad `h` y devuelva la nueva cola.
3. Escribir una función
   `pq_remove(h: np.ndarray)-> Tuple[int, np.ndarray]`
   que elimine el elemento con el menor valor de prioridad de `h` y devuelva dicho elemento y la nueva cola.

II-C. Ordenando con min heaps

Sabemos que en un Min Heap el elemento más pequeño está en su raíz. Esto sugiere el siguiente procedimiento para ordenar un array `h` :

1. Crear un min heap sobre `h` de manera in place.
2. Repetidamente:
   a) Extraer (por ejemplo, a una lista) el elemento en la raíz.
   b) Copiar el último elemento en ese momento del array que contiene el min heap a su primera posición.
   c) Rehacer el min heap aplicando heapify desde la raíz y sobre un array reducido donde se ignora el anterior último elemento.

Esto es, al aplicar heapify hay que ir reduciendo la parte del array que se considera. Por ejemplo, si en un momento dado el array `h` tiene `k` elementos, la llamada a `min_heapify` tras extraer la raíz y poner el último elemento en la raíz debe ser `min_heapify(h[ : k], 0)` , y así sucesivamente.

1. Escribir una función
   `min_heap_sort(h: np.ndarray)-> np.ndarray`
   que implemente la idea anterior devolviendo un array con una ordenación del array inicial `h` .

II-D. Cuestiones

1. Analizar visualmente los tiempos de ejecución de nuestra función de creación in place de min heaps. Establecer razonadamente a qué función f se deberían ajustar dichos tiempos.
2. Dar razonadamente cuál debería ser el coste de nuestra función de ordenación mediante Min Heaps en función del tamaño del array.
3. Analizar visualmente los tiempos de ejecución de nuestra función de ordenación mediante Min Heaps y comprobar que los mismos se ajustan a la función de coste del punto anterior.

III. Material a entregar y corrección

III-A. Material a entregar

Crear una carpeta con el nombre `p1NN` donde `NN` indica el número de pareja y añadir en ella **sólo** los siguientes archivos:

1. Un archivo Python `p1NN.py` con el código de las funciones desarrolladas en la práctica (no los scripts de medida de tiempo o dibujo de curvas) así como los `imports` estrictamente necesarios, a saber:

   ```
   import numpy as np

   from typing import Tuple
   ```

   Los nombres y parámetros de las funciones definidas en ellos deben ajustarse EXACTAMENTE a los utilizados en este documento.
   Además, todas las definiciones de funciones deben incorporar los type hints adecuados.
2. Un fichero `p1NN.html` con el resultado de aplicar el comando `pdoc` del paquete `pdoc3` al módulo Python `p1NN.py` .
3. Un archivo `p1NN.pdf` con una breve memoria que contenga las respuestas a las cuestiones de la práctica en formato pdf. En la memoria se identificará claramente el nombre de los estudiantes y el número de pareja. Si se añaden figuras o gráficos, deben tener título y nombres en los ejes y **ESTAR SOBRE UN FONDO BLANCO.**
   Se recomienda generar las gráficas usando notebooks y guardándolas como pdfs.

Una vez que se hayan puesto estos archivos, comprimir esta carpeta en un archivo llamado `p1NN.zip` o `p1NN.7z` y subirlas al buzón Moodle que se indique. **No añadir ninguna estructura de subdirectorios a la carpeta.**
La práctica no se corregirá hasta que el envío siga esta estructura.

III-B. Corrección

La corrección se hará en base a los siguientes elementos:

- La ejecución de un script que importará el módulo `p1NN.py` y comprobará la corrección de su código.
  La práctica no se corregirá mientras este script no se ejecute correctamente, penalizando las segundas presentaciones por esta causa.
- La revisión de la documentación del código contenida en los ficheros html generados por `pdoc`. **En particular, las docstrings deben ser escritas muy cuidadosamente.**
  Además, el código Python debe estar formateado según el estándar PEP-8. Utilizar para ello un formateador como `autopep8` o `black`.
- La revisión de una selección de las funciones de Python contenidas en el módulo `p1NN.py` , donde se tendrá en cuenta la claridad y calidad del código, su buen formateo y su documentación interna (en particular, comentarios y nombres de variables).
- La revisión de la memoria con las respuestas a las preguntas anteriores.