Algoritmos y estructura de datos avanzadas. Año Académico 2025-26

Práctica 1

Python básico. Medida de tiempo de ejecución. Heap

Fecha de entrega: Grupo 126: **13 de Octubre 2025** antes de la clase
Grupo 127: **14 de Octubre 2025** antes de la clase

¡Atención! Muchas de las funciones de esta práctica (y de las demás) se pasarán por un script de corrección automática. Esto quiere decir que tenéis que escribir escupulosamente las indicaciones en lo que se refiere a nombres de funciones, parámetros, valores de retorno, ficheros, etc. Las funciones no deben escribir nada en pantalla (si ponéis algunos `print` para hacer *debuging* que no se os olvide de eliminarlos antes de entregar). Los ficheros que se entregan deben contener sólo las funciones, ningún *script* de prueba ni nada que se ejecute cuando se hace el `import` del fichero.

No seguir estas indicaciones puede resultar en una penalización o, en los casos peores, en la no corrección de la práctica.

I. Python básico y tiempo de ejecución

I.A Midiendo tiempos de ejecución

Para medir el tiempo de ejecución, la manera más directa es utilizar la función `time` de la librería `time` de `Python`. La medición correcta necesita ciertas precauciones, como explicado en la apéndix de esta práctica. En este apartado utilizaremos el método allí especificado para medir el tiempo de ejecución de algunas funciones que definiremos en este mismo apartado.

I.A.1 Escribir una función:

```
time_measure(f, dataprep, Nlst, Nrep=1000, Nstat=100)
```

Con los siguientes parámetros:

`f:` una función de un parámetro cuyo tiempo de ejecución queremos medir

`dataprep:` función de preparación de datos. La función recibe un número entero `n` y devuelve una estructura de "dimensión `n`" que es aceptada por `f` como parámetro.

`Nlist:` una lista de valores enteros. El tiempo de ejecución de la función se medirá por cada uno de estos valores de `n`.

`Nrep:` número de repeticiones para cada medida elemental del tiempo de ejecución (véase en apéndix)

`Nstat:` número de repeticiones con entradas distintas de la misma dimensión para la valoración estadística (véase en apéndix)

La función devuelve una lista de tipo

[(m0,v0), (m1, v1), ..., (mk-1, vk-1)]

con `k=len(Nlist)`. El elemento `(mi,vi)` contiene la media y la varianza del tiempo de ejecución de `f` con datos de dimensión `n`

I.A.2 Escribir una función:

```
search_all(lst, v)
```

que, dada una lista de enteros (`lst`) y un valor `v`, devuelve una lista con los índices de todas las ocurrencias de `v` en `lst` (vacía si no hay coincidencias).

I.A.3 Implementa una función:

```
majority_element(lst)
```

que reciba una lista de enteros y devuelve el valor que aparece más de la mitad de las veces (o `None` si no existe tal valor).

El algoritmo debe tener complejidad $O(n)$ y utilizar solo $O(1)$ en el espacio adicional aparte de las variables auxiliares necesarias. Es decir, el algoritmo debe recorrer toda la lista una vez y resolver el problema usando solo unas pocas variables, sin memoria extra proporcional al tamaño de los datos.

I.A.4 Medición del rendimiento:

Utiliza la función time_measure para analizar el tiempo de ejecución de majority_element para listas de longitudes de 10 a 10000 en pasos adecuados. Mide tanto casos donde sí hay elemento mayoritario como donde no. ¿Cuál es el comportamiento de las función? ¿Cuál es la complejidad? Justifica y razona cada respuesta.

* * *

I.B Algoritmos de ordenación: comparativa empírica

En este apartado vamos a implementar y medir el tiempo de ejecución de dos funciones de ordenación.

I.B.1 Escribir la función:

```
bubble_sort(lst, orden)
```

que recibe una lista de enteros aleatoria `lst`, y debe ordenarla de menor a mayor si `orden` tiene valor `True`, en caso contrario, de mayor a menor.

I.B.2 Escribir la función:

```
merge_sort(lst, orden)
```

con las mismas especificaciones de la función del apartado anterior.

I.B.3 Usar la función `time_measure` para analizar el tiempo de ejecución de estas dos funciones sobre listas aleatorias de diferentes tamaños. Los valores de `lst` en que efectuar la medida se eligirán de manera tal que se pueda ver el comportamiento de las dos funciones de manera clara, teniendo en cuenta el caso mejor y peor de cada una. ¿Cuál es el comportamiento de las funciones? ¿Cuál es la complejidad? Justifica y razona cada respuesta.

II. Heap

II.A Max Heap en listas de `Python`

En este apartado crearemos las funciones básicas para trabajar con max heap utilizando las listas de `Python`.

II.A.1 Escribir la función

```
h = heap_heapify(h, i)
```

que recibe una lista `h` conteniente un *heap* que hay que arreglar, y un índice `i` dentro de la cadena con el índice del elemento de donde se empieza el *heapify*. Implementar la función de manera recursiva.

La función cambia el *heap* `h` "in place" y devuelve el mismo h.

II.A.2 Escribir la función

```
h = heap_insert(h, key)
```

que recibe la lista `h` conteniente un *heap* (se asume que la lista contiene un *heap* correcto: no hace falta comprobar la corrección), inserta la clave `key` en el *heap* y devuelve el *heap* con la clave insertada. La función cambia el parámetro `h`.

II.A.3 Escribir la función

```
(h, e) = heap_extract(h)
```

que elimina el elemento mayor del *heap* `h` (arreglando el *heap*) y lo devuelve en la variable `e`. El parametro `h` es modificado y es devuelto como primer elemento de la tupla `(h,e)`.

II.A.3 Escribir la función

```
h = heap_create(h)
```

que recibe una lista desordenada `h`, la transforma, "in place" en un *heap*, y la devuelve.

II.B Colas de prioridad sobre Max Heap

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

II.B.1 Escribir la función

```
h = pq_ini()
```

que inicialice una cola de prioridad vacía y la devuelva.

II.B.2 Escribir la función

```
h = pq_insert(h, key)
```

que inserte el elemento `key` en la cola de prioridad `h` y devuelva la nueva cola con el elemento insertado.

II.B.3 Escribir la función

```
(h, e) = pq_extract(h)
```

remueva el elemento con mayor prioridad de la cola y devuelve el elemento y la nueva cola.

III. ¿Qué tengo que entregar?

i) Un fichero `p1.py` que contenga:

`i.1) Las funciones solicitadas en el apartado I.A (I.A.1, 1.A.2 y 1.A.3).`

`i.2) Las funciones del apartado II (heap y colas de prioridad).`

¡Atención! El fichero debe contener sólo las funciones, ningún script de prueba.

ii) Un fichero `measure.pdf` con las gráficas del tiempo de ejecución de la función del apartado I.A y de las funciones del apartado I.B así como un análisis de las mismas.