Universidad Autónoma de Madrid
Escuela politécnica Superior
Grado en Ingeniería biomédica

Algoritmos y Estructuras de Datos

2021-22

Práctica 1

Fecha de Asignación	16 Septiembre 2021
Fecha de Entrega	30 Septiembre 2021

Información sobre la entrega de la práctica

Las prácticas se entregarán en un sólo fichero creado utilizando `tar` o `zip` (**no** utilizar `rar`). El fichero contendrá todos los ficheros que en cada práctica se pide entregar, bien organizados en carpetas. El nombre del fichero debe adecuarse al formato siguiente

`ALGED-2021-<N. pareja>-<Apellido1.1>-<Apellido2.1>.{tar.gz|zip}`

Por ejemplo, la pareja 03 formada por Carmen García y Juan Español entregarán el archivo siguiente:

`ALGED-2021-03-Garcia-Español.zip`

Es importante que el nombre siga este formato para facilitar la identificación de los autores y por tanto la corrección. Es importante también que **todos** los ficheros que entregáis contengan el nombre del autor o de los autores. Esto se aplica no sólo a las memoria sino también al código. Los nombres de los autores deben ser claramente indicados en la cabecera de cada fichero, que debe tener este formato:

```
################################################################
#
#   Algoritmos y Estructura de datos 2021-22
#
#   Práctica: 1
#   Fichero:  <nombre del fichero>
#   Autor:    <nombre de los autores o autor>
#   Fecha:    <fecha en que se ha completado>
#
#   <Descripción del contenido del fichero>
#
################################################################
```

La memoria se debe entregar **en formato pdf**, ningún otro formato es aceptado. Memorias entregadas en otros formatos serán ignoradas y la puntuación se reducirá correspondientemente.

En esta práctica empezaremos a ver cómo medir el tiempo de ejecución de un algoritmo usando los métodos ilustrados en los apuntes de prácticas. Mediremos el tiempo de ejecución de dos clases de funciones: las funciones de listas y las funciones de multiplicación de matrices de la librería `numpy`.

Operaciones de listas

Las listas de `python` son muy completas y permiten muchas operaciones. Por un lado, se trata de listas sin tipos, es decir, cualquier elemento de la lista puede ser de un tipo diferente. Por ejemplo, una lista como

```
>>> lst = [1, 3.8, 'hello', [1, 2, 'world']]
>>>
```

es perfectamente legal en `python`, mientras no lo es en otros lenguajes, dado que los cuatro elementos de la lista son de tipos diferentes (el primero es un entero, el segundo un número *floating point*, el tercero un texto y el cuarto una lista. Esta es una característica típica de las listas enlazadas, mientras las cadenas con acceso directo suponen que todos los elementos sean del mismo tipo (para el acceso directo, cada elemento tiene que tener el mismo tamaño). Si embargo, las listas

de `python` permiten algo que se parece al acceso directo a los elementos. En la lista anterior, por ejemplo, podemos hacer

```
>>> lst[3]
[1, 2, 'world']
>>>
```

En esta primera parte analizaremos el comportamiento de varias funciones de lista para averiguar si son características de listas enlazadas, de listas doblemente enlazadas, o si los diseñadores de `python` han encontrado alguna manera de acelerar todas las operaciones. Mediremos cuatro tipos de operaciones:

i) Añadir un elemento al final de la lista.

ii) Añadir un elemento al principio de la lista.

iii) Añadir un elemento en un punto dado de la lista.

iv) Aceder a un elemento en un punto dado de la lista.

Antes de empezar una medida, es deseable hacerse una idea de lo que debería pasar creando un modelo (mental o físico) de la situación. De esta manera, si las medidas confirman nuestras intuiciones, tendremos más confianza en nuestro modelo[1], mientras si no las confirman, invalidarán nuestro modelo y nos darán indicaciones sobre lo que está pasando de verdad. Los modelos elementales de listas enlazadas son dos: la *lista enlazada sencilla* y la *lista doblemente enlazada*. Estas dos posibilidades se representan en la Figura 1. Consideremos la primera solución **(a)**. En una lista enlazada sencilla lo que esperamos es que la inserción al final de una lista con n elementos tome un tiempo $\Theta(n)$, dado que hay que recorrer toda la lista siguiendo los punteros. La inserción en la cabeza de la lista debería tomar un tiempo $\Theta(1)$, dado que se hace simplemente cambiando un par de punteros, y el acceso al elemento k debería tomar un tiempo $\Theta(k)$. Lo mismo vale para insertar un elemento en posición k: hay que acceder a la posición k y luego arreglar unos punteros, por tanto el tiempo necesario es $\Theta(k)$. En nuestro caso, $k = \lfloor n/2 \rfloor$, por tanto esperamos un tiempo $\Theta(\lfloor n/2 \rfloor) = \Theta(n)$.

En el caso de la solución **(b)**, la lista doblemente enlazada, hay varias diferencias. El tiempo para insertar un elemento en la cabecera no cambia, sigue siendo $\Theta(1)$. Por otro lado ahora tenemos acceso directo a la cola de la lista, por tanto esperamos un tiempo de inserción en la cola de la

[1] Ninguna medida puede confirmar un modelo: sea cual sea el resultado, en principio hay muchos modelos que lo pueden explicar. Por otro lado, una medida que no se ajusta a las previsiones del modelo que tenemos es suficiente para invalidar el modelo. Estamos en presencia de un ejemplo del principio de *falsificabilidad* de las teoría científicas, originalmente enunciado por Karl Popper.

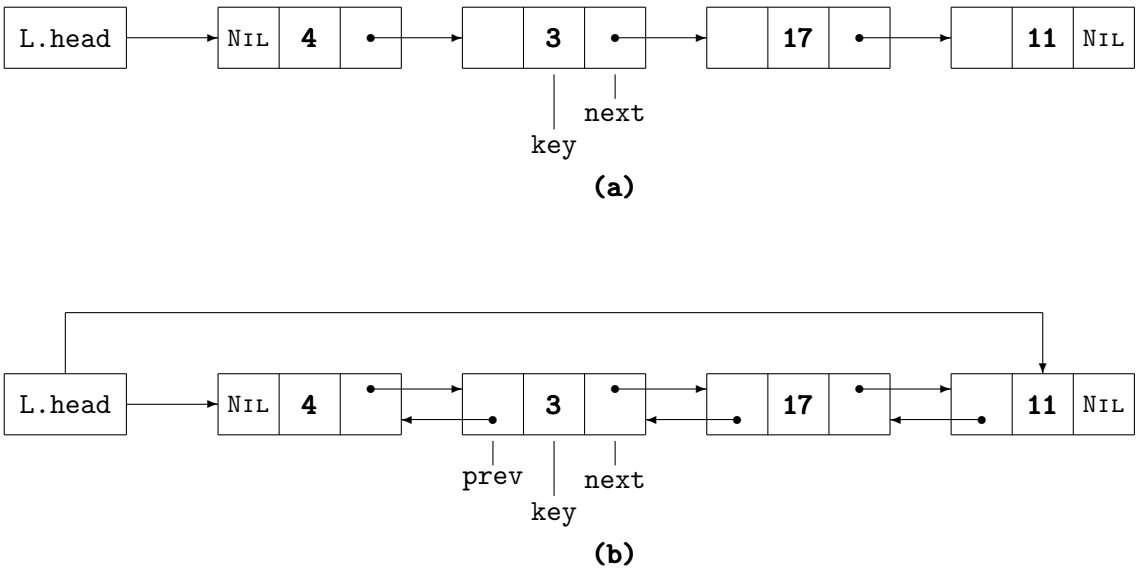

Figure 1: Las dos organizaciones básicas de una lista. En la *lista simplemente enlazada* (a) cada elemento contiene un puntero al elemento siguiente, y la cabecera de la lista (`head`) contiene un puntero al primer elemento de la lista. En la *lista doblemente enlazada* cada elemento contiene un puntero al elemento siguiente y al precedente. La cabecera contiene un puntero al primer elemento de la lista y uno al último.

lista $\Theta(1)$. El acceso al elemento k, ya sea para leerlo ya sea para insertar un elemento, es más complicado, en cuanto tenemos dos maneras de llegar al elemento k: desde el principio de la cola o desde el final. Es posible que un sistema bien optimizado acceda al elemento k desde el principio si $k < n/2$ y desde el final si $k > n/2$. En este caso el tiempo de acceso sería $\min(k, n-k)$.

Naturalmente es posible que todas estas observaciones se reduzcan a nada, que los diseñadores de `python` hayan usado una estructura completamente diferente. Esto es lo que intentamos averiguar en la primera parte de esta práctica. Para esto se pide escribir funciones que efectúen las siguientes medidas:

i) Escribid una función que genere una lista con n elementos aleatorios (enteros o números *floating point*, da igual) y añada un elemento al final de la lista (es decir, que ejecute, sobre la lista `lst`, la operación

`lst = lst + [x]`,

donde `x` es un elemento aleatorio a insertar. Repetir la operación un número suficiente de veces como para conseguir una medida de tiempo fiable, y repetir todo el proceso un número

suficiente de veces como para medir media y varianza. Crear una gráfica que ilustre el tiempo de ejecución de la operación en función de n, siendo $n = 1000, 2000, \ldots, 100000$. Es posible que en estas medidas consigáis varianza cero. ¿Es normal? ¿Por qué? Comentad el tema en la memoria.

ii) Escribid una función que genere una lista con n elementos aleatorios (enteros o números *floating point*, da igual) y añada un elemento al comienzo de la lista. Repetir las medidas y las consideraciones del punto i).

iii) Escribid una función que genere una lista con n elementos aleatorios (enteros o números *floating point*, da igual) y añada un elemento en el medio de la lista, es decir

```
n = len(lst)

lst = lst[:int(n/2)] + [x] + lst[int(n/2):]
```

Repetir las medidas y las consideraciones del punto i).

iv) Escribid una función que genere una lista con n elementos y acceda al elemento número k. Medir el tiempo como en los apartados precedentes esta vez en función de k, siendo $k = 1000, 2000, \ldots, 100000$.

Nótese que esta vez la lista se crea una sola vez con un valor de n suficientemente grande para efectuar todas las medidas. Nos interesa saber si el sistema usa algún truco para accelerar el acceso en ciertas posiciones de la lista, por tanto crearemos la lista de manera tal que accederemos, durante la medida, a todos sus elementos, es decir utilizaremos una lista de longitud $n = 2k_{max}$, donde k_{max} es el valor de k más alto que se usa. En este caso, $n = 100.000$.

Cread las gráfica con el comportamiento de las funciones de listas en todos estos escenarios y comentadla: ¿Son los resultados compatibles con uno de los modelos de la Figura 1 o hay sorpresas? (Sugerencia: es probable que haya sorpresas. A mi, por ejemplo, con mi versión de `python` me han salido resultados que no esperaba.)

Recordamos que el esquema que se usa para efectuar este tipo de medidas es el siguiente:

```
Nrep = # Numero de repeticiones para la medida del tiempo
Nstat = # Numero de repeticiones para la estadistica
res = Nstat*[0.0]

for each n:
     for trial in range(Nstat):
       data = # datos aleatorios para el problema
       t1 = time()
       for t in range(Nrep):
         ejecuta(data) # Ejecucion de la funcion a medir
       t2 = time()
       res[trial] = float(t2-t1)/float(Nrep)
     mean = average(res)
     var = variance(res)
     output los valores para este n
```

Multiplicación de Matrices

En esta segunda parte de la práctica haremos medidas parecidas a las de la primera parte, con las mismas modalidades, pero en este caso la función que evaluaremos es la multiplicación de matrices que forma parte de la librería `numpy` de `python`. En el caso de una multiplicación de matrices puede haber alguna ambigüedad a la hora de establacer el parametro n respecto al que se evalúa el tiempo de ejecución. Si consideramos dos matrices $M, N \in \mathbb{R}^{n\times n}$ podemos considerar como "dimensión del problema" el número n (la dimensión del espacio lineal en que operan las matrices) o el número de elementos $p = n^2$ de cada una de las dos matrices. Tradicionalmente, el parámetro que se usa es n, y este es el parámetro que utilizaremos por el momento.

La estructura del código es la misma que se ha usado en la primera parte. En este caso, la generación de datos se hará creando una estructura `array` según el estándar de `numpy`

```
M = array( [ [random.uniform(-1,1) for _ in range(n)] for _ in range(n)] )

N = array( [ [random.uniform(-1,1) for _ in range(n)] for _ in range(n)] )
```

El resto del código es esencialmente igual, excepto por tener que remplazar la multiplicación de las dos matrices en lugar de las operaciones de lista.

En esta parte se pide lo siguiente:

i) Efectuar la medida del tiempo de ejecución de la multiplicación (media y varianza) por n entre 50 y 10.000 en pasos de 100. ¿Cómo es la varianza? ¿Por qué? Crear una gráfica.

ii) La multiplicación de matrices es (en tiempo) polinomial, $\Theta(n^q)$. Usad los resultados para estimar q.

iii) ¿Que cambia si en lugar de usar n como dimensión del problema usamos el número de elementos $p = n^2$? ¿Cambia el exponente q que habéis estimado al paso precedente? Si la respuesta es afirmativa, ¿Que valor asume? Si la respuesta es negativa, ¿Por qué no cambia?

Entrega y puntuación

En el fichero de entrega, cuyo nombre tiene que respetar el formato explicado anteriormente, debe haber lo siguiente:

i) Una carpeta `listas` con todo el código utilizado para efectuar las medidas sobre las funciones de lista (primera parte).

ii) Una carpeta `matrix` con todo el código utilizado para efectuar las medidas sobre multiplicación de matrices (segunda parte).

iii) En la carpeta principal, una memoria **en formato pdf** (no se aceptarán otros formatos) con todas las gráficas que se han pedido, una discusión (breve) de la impementación de los algoritmos y una discusión completa de los resultados.

Criterios de puntuación:

Tarea	puntos
Código de pruebas de las listas	2
Resultados y discusión de las listas	2
Código de pruebas de las matrices	2
Resultados y discusión de las matrices	2
Estilo de código (comentarios, etc.)	2