Algoritmos y estructura de datos avanzadas. Año Académico 2024-25

Práctica 1

Python básico. Medida de tiempo de ejecución. Heap

Fecha de entrega: Grupo 126: **7 de Octubre 2024** antes de la clase
Grupo 127: **8 de Octubre 2024** 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:

```
two_sum(lst, n)
```

que, dada una lista de enteros (`lst`) y un valor `n`, devuelve `True` si en la lista hay dos enteros que dan como suma `n`, y devuelve `False` en caso contrario.

I.A.3 Utilizar la función definida en I.A.1 para medir el tiempo de ejecución de la función definida en I.A.2 para valores de `n` de 10 a 10000 en pasos de 100. ¿Cuál es el comportamiento de la función? ¿Es posible una implementación que tenga complejidad $O(n)$? (Sugerencia: intentad con un diccionario utilizado de manera oportuna).

¡Ojo!: por definición la función `f` debe recibir **un** parámetro. Esto puede suponer tener que empaquetar parámetros en una tupla. Consideremos una simple función de búsqueda lineal:

```
def search(lst, v):
     for u in lst:
       if u == v:
         return True
     return False
```

El problema es que esta función necesita **dos** parámetros. Así no puede funcionar[1]. Hay que transformarla en una función de un parámetro. En este caso la solución es sencilla: el parámetro será una tupla con dos elementos: el primero es la lista en que buscamos, el segundo el elemento que buscamos.

```
def search(par):
     lst = par[0]
     v = par[1]
     for u in lst:
       if u == v:
         return True
     return False
```

La función de creación de datos, claramente, tiene que respetar este formato:

```
def search_data(n):
     lst = [random.uniform(0,1) for _ in range(n)]
     idx = random.randint(0, n-1)
     return (lst, lst[idx])
```

[1] Curiosidad: la búsqueda lineal en `Python` se puede hacer en una sola línea:

```
def search(lst, v):
     return sum([1 for x in lst if x == v])>0
```

Ejercicio: intentar entender como funciona. ¿Qué tiempo de ejecución tiene? ¿Hay diferencia en esta implementación entre caso mejor y caso peor?

* * *

I.B Búsqueda binaria en listas
En este apartado vamos a implementar y medir el tiempo de ejecución de funciones de búsqueda binaria sobre listas. El ejercicio tiene su interés: sabemos que el tiempo de ejecución de una búsqueda binaria en una lista de n elementos ed $O(\log n)$ *si el acceso a un elemento arbitrario de la lista es* $O(1)$. Este ejercicio nos permitirá analizar la eficiencia del acceso a elementos de una lista en `Python`: si el resultado del análisis del tiempo de ejecución será $O(\log n)$ el acceso a lista es eficiente, en caso contario no lo es.

I.B.1 Escribir la función **recursiva**:

```
rec_bs(lst, lft, rgt, key)
```

que recibe una lista ordenada de enteros `lst`, dos índices dentro de la lista, `lft`, `rgt` y un entero `key` y busca un elemento con valor igual a `key` en la porción de lista `lst[lft:rgt]`. La función devuelverá el índice `idx` tal que `lst[idx]==key` o `None` si `key` no aparece en la lista `lst[lft:rgt]`.

I.B.2 Escribir la función **iterativa**:

```
itr_bs(lst, lft, rgt, key)
```

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. Los valores de `n` en que efectuar la medida y los demás parámetros se eligirán de manera tal que se pueda ver el comportamiento de las dos funciones de manera clara. Se elija la clave `key` de manera tal que las funciones se ejecuten en el caso peor. ¿Cuál es el comportamiento de las funciones? ¿Parece logarítmico? ¿Cómo se puede crear una gráfica que enseñe claramente si el comportamiento es logarítmico? Si no lo es, ¿qué comportamento tiene? ¿Qué conclusiones se pueden derivar sobre la indexación de la listas en `Python`? Dado que se ha elegido `key` para conseguir siempre el caso peor, la variancia de la medida es, en teoría, cero. ¿Es este el caso?

II. Heap

II.A Min Heap en listas de `Python`
En este apartado crearemos las funciones básicas para trabajar con min 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`
dqtin 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 asuma 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 menor 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 Min Heap

Vamos a usar las funciones anteriores sobre *min 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 menor salen antes que los que tienen prioridad mayor.

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 menor prioridad de la cola y devuelve el elemento y la nueva cola.

III. ¿Qué tengo que entregar?

i) Un fichero `measure.py` con las funciones de medición (apartado I.A.1) y las funciones del apartado I.A.2 y I.B. El fichero debe contener sólo las funciones, ningún script de prueba.

ii) Un fichero `measure.pdf` con la gráfica del tiempo de ejecución de la funciones del apartado I.A y un análisis del mismo. Si se han ejecutado mejoras al algoritmo, estas también se deben comentar aquí.

iii) Un fichero `heap.py` con las funciones del apartado II (heap y colas de prioridad). El fichero debe contener sólo las funciones, ningún script de prueba.

APÉNDIX:
MEDIDA DEL TIEMPO DE EJECUCIÓN

El análisis teórico de los algoritmo es, siempre, el que nos da más información sobre su eficiencia y, a menudo, sobre como mejorarla. Pero hay casos, sobre todo en el caso de sistemas complejos en que varios algoritmos interactúan en una manera difícil de prever, en que el análisis es imposible o poco informativo. En estos casos, es posible estimar el comportamiento de un algoritmo experimentalmente. El principio de esta medida es muy sencillo, pero la realización de una medida correcta y fiable supone ciertas precauciones de que hablaremos en esta sección.

1 Preliminares de medida

El análisis teórico del tiempo de ejecución de los algoritmos se ha hecho usando una medida abstracta, es decir, el número de operaciones elementales que se ejecutan. La justificación para esta solución era abstraer la medida de los detalles del ordenador en que se ejecuta el algoritmo. En el caso de la medida experimental es posible utilizar la misma abstración utilizando oportunos programas (**profilers**) que miden cuantas veces se ejecuta cada línea de programa. Esto no resuelve completamente el problema, en cuanto programas escritos en lenguajes diferentes tendrán un número diferente de líneas de programa, y habrá que analizar que líneas "cuentan" para la evaluación.

Aquí no consideraremos este tipo de medida, sino que consideraremos la medida directa del tiempo de ejecución del algoritmo. Esta solución es válida siempre y cuando recordemos que lo que nos interesa es el comportamiento del algoritmo cuando la dimensión de los datos crece, y no los valores efectivos de tiempo, que dependen del específico ordenador en que lo ejecutamos y también del lenguaje de programación que usamos para implementar el algoritmo. Lo que nos interesa es derivar una curva $T(n)$ para varios valores de n y ver cual es su comportamiento cuando n crece. Los valores exactos de esta función dependen de muchos factores externos al algoritmo y en general no nos interesan.

Un primer problema que se nos presenta es cómo implementar el algoritmo. Hay lenguajes de programación que permiten "trucos" u optimizaciones que pueden aumentar la velocidad de ejecución. Es cierto que rara vez estos trucos permiten alterar el comportamiento asintótico del algoritmo, pero siempre tenemos que plantearnos el problema de si y en que medida sea correcto usar estos trucos. Hay que considerar que ciertas posibilidades de aumentar la velocidad son muy dependientes del lenguaje de programaciń usado y por tanto los resultados obtenidos pueden no ser generalizable a todos los lenguajes. Mucho depende de lo que queremos determinar: ¿Queremos saber como se comporta el algoritmo abstracto, de manera más o menos independiente de su implementación, o queremos saber como se comporta una implementación específica? En este último caso podemos (y debemos) utilizar todos los instrumentos y los trucos que el sistema nos pone a disposición, pero los resultados que obtendremos no serán generales y no nos darán ninguna indicación de como el algoritmo se comportará si implementado en otro lenguaje o en otro sistema operativo.

Es es el caso más común en las aplicaciones: se implementa un algoritmo usando un lenguaje específico, y queremos medir sus prestaciones para saber como integrarlo en un sistema complejo y en que medida nuestro algoritmo condiciona las prestaciones de todo el sistema. En este caso puede ser útil optimizar el algoritmo.

El primer caso se presenta, por ejemplo, cuando desarrollamos un nuevo algoritmo y nos interesa saber como se comporta independientemente del lenguaje en que se implementa. En este caso es aconsejable hacer una implementación lo más directa posible del algoritmo, sin utilizar trucos específico de un lenguaje de programación.

1.1 Medida básica

Casi todos los lemguajes de programación tienen funciones para medir la hora corriente en un formato que permite con gran facilidad la determinación de intervalos. La mayoría de los lenguajes usan el estandar UNIX, y miden el tiempo como la cantidad de milisegundos que han transcuridos desde la medianoche del 1 de enero de 1970. La medida básica para una entrada de tamaño n se puede esquematizar con el siguiente código[2]:

```
input = generate_random_input(n)
t1 = current_time()
_ = algorithm(input)
t2 = current_time()
print (t2 - t1)
```

Aquí la función "generate_random_input" genera una entrada de tamaño n para el algoritmo y la función "algorithm(input)" implementa el algoritmo que estamos evaluando. La función "generate_random_input" debe ser diseñada con cuidado, en cuanto la distribución de probabilidad de los datos generados no debe polarizar de ninguna manera la medida.

Example I:
Consideremos el ejemplo de la búsqueda lineal. En este caso la función "generate_random_input" debe generar dos datos: una cadena de longitud n y un valor que hay que buscar. Una posibilidad es la siguiente:

```
def generate_random_input(n)
1.   data = []
2.   for i in range(n):
3.     data += [random_int(-MAX,MAX)]
4.   idx = random_int(0,n-1)
5.   return (data, data[idx])
```

La función devuelve una cadena de enteros aleatorio (dentro de un rango oportuno definido por el parametro de sistema MAX) y un elemento aleatorio de la cadena como valor a buscar.

Hay un pequeño problema aquí: los datos generados serán siempre relativos al caso en que el valor buscado se encuentra efectivamente en la cadena. Es decir, nunca nos encontraremos en el caso peor en que el algoritmo tiene que recorrer toda la cadena sin encontrar nada. En circunstancias normales este no es un problema: lo que nos interesa normalmente es verificar que el algoritmo tiene efectivamente un tiempo de ejecución $O(n)$, y esto se puede hacer sin recurrir al caso peor (en el caso en que el dato esté presente en la cadena, el tiempo medio de ejecución es $\sim n/2$ y el peor $\sim n$).

En algunos casos es posible que tengamos información más específica sobre el entorno en que funcionará nuestro algoritmo. Por ejemplo, podemos saber que existe una probabilidad p que el valor buscado no esté en la cadena. Si es así, podemos generar datos más adecuados para el problema con la función:

[2] En este fragmanto aparece la variable "dummy" `_` de `python`. Es una variable que se pone en sitios donde se necesita una variable pero no nos interesa el valor que se le va a asignar. Es una variable de sola escritura: se le puede asignar un valor pero este valor desaparece y ya no se puede leer.

```
def generate_random_input(n, p)
1.   data = []
2.   for i in range(n):
3.     data += [random_int(-MAX,MAX)]
4.   idx = random_int(0,n-1)
5.   w = random(0,1)
6.   if w < p:
7.     v = max(data) + 1
8.   else:
9.     v = data[idx]
10.  return (data, v)
```

Sin embargo, es importante entender que en este caso lo que estamos midiendo no son las prestaciones del algoritmo en abstracto, sino sus prestaciones en una situacióin específica, prestaciones que no se pueden generalizar a otros casos y a otras situaciones (si bien, en este ejemplo, la presencia de p no cambia el comportamiento lineal del algoritmo).

Notamos que en este caso, si $p = 1$, es decir, si siempre generamos un valor v que no está en la cadena, conseguimos la evaluación de caso peor del algoritmo. Esta puede ser una estrategia de medida muy útil, pero depende de si el análisis teórico del algoritmo nos permite generar datos que representan su caso peor. No siempre esto es posible: si el análisis teórico no nos permite saber como generar datos de caso peor, será necesario generar datos aleatorios y utilizar un análisis estadístico para estimar el comportamiento de caso peor.
(end of example)

Este esquema sencillo tiene varios problemas. El principal es que el tiempo de ejecución del algoritmo es muchas veces demasiado corto para permitir una buena medida: si el tiempo de ejecución es del orden del milisegundo, la granularidad de la medición del tiempo (1ms) no será suficiente para conseguir una medida razonable. En este caso es neceario repetir la ejecución del algoritmo un gran número de veces para conseguir un tiempo total de ejecución suficientemente largo como para poderlo medir con buena precisión. El esquema es el siguiente

```
Nrep = 1000
input = generate_random_input(n)
t1 = current_time()
for i in range(Nrep)
  _ = algorithm(input)
t2 = current_time()
texe = float(t2 - t1)/float(Nrep)
```

Este esquema supone un cargo adicional: el tiempo necesario para ejecutar el bucle for. En la casi totalidad de casos este tiempo es mucho inferior al tiempo necesario para ejecutar el algoritmo, y se puede despreciar. En caso contrario, es necesario estimarlo y restarlo al tiempo total de ejecución:

```
Nrep = 10000
t1 = current_time()
for i in range(Nrep):
  pass
t2 = current_time()
tover = float(t2 - t1)
input = generate_random_input(n)
t1 = current_time()
for i in range(Nrep):
  _ = algorithm(input)
t2 =current_time()
texe = float(t2 - t1 - tover)/float(Nrep)
```

Ignorando este problema, podemos calcular el comportamiento del algoritmo por varios valores del tamaño del ingreso con el simple programa

```
Nrep = 1000
res = []
nvals = [n_1, ..., n_m]
for n in nvals:
  input = generate_random_input(n)
  t1 = current_time()
  for i in range(Nrep):
    _ = algorithm(input)
  t2 = current_time()
  res += [float(t2 - t1)/float(Nrep)]
```

Todavía tenemos un problema: por cada valor de n, generamos una sola entrada y calculamos el tiempo de ejecución del algoritmo sólo con esa entrada. Es posible que esta entrada no sea representativa del caso peor o del caso medio de ejecución y por tanto, es posible que nos de resultados poco fiables. Por ejemplo, en el caso de la funcióm del Ejemplo 1, si por casualidad el valor idx generado en la línea 5 es 0, el valor a buscar, v, será el primer valor de la cadena, y el algoritmo lo encontrará inmediatamente, falseando el comportamiento medio y de caso peor para ese valor.

Una medida estadísticamente más fiable se consigue repitiendo la medida con varios ingresos generados aleatoriamente. Con el conjunto de tiempos de ejecución así obtenidos podemos estimar el tiempo medio de ejecución, su varianza, y el tiempo máximo.

```
Nrep = 1000
Nstat = 100
res = []
nvals ← [n_1, ..., n_m]
for n in nvals:
  partial = []
  for s in range(Nstat):
    input = generate_random_input(n)
    t1 = current_time()
    for i in range(Nrep):
      _ = algorithm(input)
    t2 = current_time()
    partial += [float(t2 - t1)/float(Nrep)]
  res += [partial]
  ave = []
  var = []
  worst = []
for k in range(len(nvals)):
  ave += [ sum(res[k])/float(Nstat) ]
  worst += [max(res[k])]
for k in range(len(nvals)):
  var += [ sum( [(res[k][u]-ave[k])**2 for u in range(Nstat)] )/float(Nstat)]
```

Con este procedimiento podemos conseguir una buena estimación del tiempo de ejecución del algoritmo para los valores buscados de n (veremos en la prx́ima sección como evaluar su calidad). Se trata de un método adecuado para casi todas as situaciones de medidas que se encuentran en la práctica. Para su aplicación es necesario, sin embargo, evitar que el programa sea interrumpido por otras actividades del ordenador. El tiempo de ejecución que medimos es el tiempo *real* (de reloj) y si en un momento dado el algoritmo es interrumpido por otro proceso, este "tiempo muerto" se añadirá al tiempo de ejecución efectivo, falsando la medida para ese valor de n. El resultado son gráficas de tiempo como la de la Figura 1. Para $n = 600$, algún proceso de sistema ha interrumpido el programa de medida, y este tiempo muerto se ha añadido al tiempo de ejecución de uno o varios intento de medida. Esto ha generado **outliers**, es decir, ejecuciones con un tiempo mucho mayor que las demás ejecuciones con el mismo n, y la presencia de estos outliers se revela en la media más grande (el pico para $n = 600$) y en la varianza mayor que en los demás casos.

Hay varias maneras de limitar o eliminar el problema. La primera, más inmediata, es no utilizar el ordenador para ningún otro programa aplicativo al mismo tiempo en que se está realizando la medida. También es oportuno asignar al programa de medida una prioridad muy alta, de manera tal que la interferencia de otros programas pueda ser limitada. Algunos sistemas operativos, tales como muchas versiones de UNIX para *workstation* permiten arrancar el sistema operativo en modalidad *single user* (sólo un usuario). En general, en esta modalidad, la funcionalidad es mínima: por ejemplo, el ordenador presenta sólo una terminal de texto y no permite el uso de interfaces gráficas. Si el sistema en que se efectua la prueba permite esta posibilidad (es necesario tener privilegios de administrador), se trata de la manera mejor de eliminar el problema mencionado.

Si esto no es posible, si es inevitable que el sistema operativo interrumpa de vez en cuando la ejecución, es posible que haya que tomar medidas. Primero, naturalmente, hay que verificar si el problema es real: ejecutando el algoritmo de medida es posible observar si en algunos casos hay varianzas insolitamente altas, que revelan el problema. Si se determina que efectivamente hay un problema, y si no es posible intervenir sobre el sistema para limitar las interferencias, una solución paliativa puede ser distribuir la interferencia sobre todos los valores de n. En lugar de ejecutar, por cada intento, todos los valores de n uno tras el otro, se generará un orden aleatorio, y por cada uno de

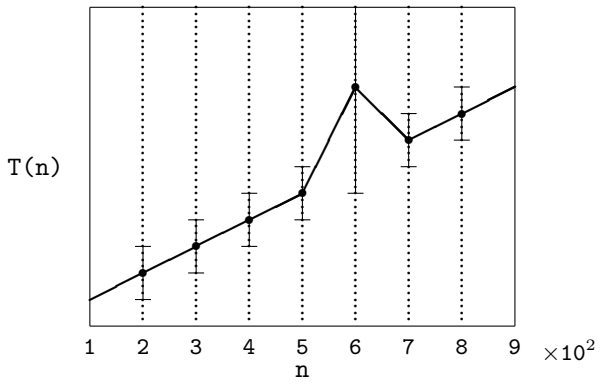

Figure 1: Problema en la medida del tiempo de ejecución causado por procesos concurrentes en el sistema operativo. En este caso, durante el análisis del caso $n = 600$, algún proceso del sistema operativo ha interrumpido nuestro programa, y el "tiempo muerto" resultante se ha añadido a una o más ejecuciones, resultando en el pico que sobresale de un comportamiento generalmente lineal. La mayor desviación estándar (la desviación estándar es la raíz cuadrada de la varianza) indica que allí hemos tenido un problema: algunas de las ejecuciones para $n = 600$ han tardado más que las otras, y estos *outliers* generan el aumento de la media y de la varianza. (El dibujo es esquemático y no refleja el resultado de pruebas efectivas)

los Nstat intentos se recorrirán los valores de n en un orden distinto:

```
Nrep = 1000
Nstat = 100
array res [m][Nstat]
nvals ← [n_1,...,n_m]
for s ← 1 to Nstat do
  lst ← scramble([1,...m])
  for k ← 1 to m do
    n ← nvals[lst[k]]
    input ← generate_random_input(n)
    t1 ← current_time()
    for i ← 1 to Nrep do
      dummy ← algorithm(input)
    od
    t2 ← current_time()
    res[lst[k]][s] ← float(t2 - t1)/float(Nrep)
  od
od
array ave [m]
array var [m]
array worst [m]
for k ← 1 to m do
  ave[k] = average(res[k])
  var[k] = variance(res[k])
  worst[k] = max(res[k])
od
```

La función *scramble* "revuelve" la lista, poniendo sus elementos en un orden aleatorio. De esta manera, por cada uno de las Nstat medidas que se usan para obtener los datos estadísticos se analizarán los valores de n en un orden diferente. Nótese que el orden de los bucle for ha cambiado respeto al metodo precedente: aquí el bucle exterior se ejecuta sobre los Nstat intentos, y el interior sobre los valores de n. En el ejemplo precedente, el orden de los bucles podía ser cambiado sin cambiar sustancialmente el método. No es así en este caso: es necesario que el bucle sobre los valores de n sea el interior.