|  |
| --- |
| Ingenieria informatica  Escuela Politécnica Superior  Universidad Autónoma De Madrid |
| ALGORITMIA |
| PRÁCTICA 1 |
|  |
| **David Teófilo Garitagoitia Romero** |
| **Daniel Cerrato Sanchez**  **Pareja 08**  **10/4/2022** |

|  |
| --- |
|  |

Índice de Contenidos

[1. Cuestiones I-C 2](#_Toc115781311)

[1. Cuestión 1 2](#_Toc115781312)

[2. Cuestión 2 3](#_Toc115781313)

[3. Cuestión 3 4](#_Toc115781314)

[2. Cuestiones II-D 5](#_Toc115781315)

[1. Cuestión 1 5](#_Toc115781316)

[2. Cuestión 2 7](#_Toc115781317)

[3. Cuestión 3 7](#_Toc115781318)

[4. Cuestión 4 8](#_Toc115781319)

# Cuestiones I-C

## Cuestión 1

**¿A qué función f se deberían ajustar los tiempos de multiplicación de la función de multiplicación de matrices?**

**Usar el Código inferior para ajustar valores de la forma *a· f(t)+b* a los tiempos de ejecución de la función que hayamos ´ guardado en el array Numpy timings , para a continuación dibujar tanto los tiempos como el ajuste calculado por la función, y comentar los resultados.**

Se deberían ajustar a una función cúbica, ya que se necesitan tres bucles para realizar el recorrido de las matrices factor:

f = O(N^3)

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Como se puede observar, el ajuste es muy bueno ya que, al coger los tiempos mejores de cada interacción, se consigue una gráfica suavizada.

Con esto se demuestra que el método matrix\_multiplication() tiene un tiempo de ejecución del orden de O(N³)

## Cuestión 2

**Calcular los tiempos de ejecución que se obtendrán usando la multiplicación de matrices a.dot(b) de Numpy y compararlos con los anteriores**

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Observando el eje vertical, el cual refleja el mejor tiempo de ejecución de cada iteración en segundos, se puede comprobar que el método de multiplicación de matrices propuesto por Numpy es muchísimo más eficaz que la función "matrix\_multiplication", donde las primeras medidas son del orden de 10^(-6) y las segundas de 10^(-3).

## Cuestión 3

**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. ¿Se puede encontrar alguna relación entre ellos?**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

Se puede observar que el método recursivo es un poco más lento que el método iterativo, esto es debido a que, aunque la complejidad temporal teórica de ambas es la misma (se puede ver en la gráfica como ambas siguen la función logarítmica), la versión recursiva debe tardar más tiempo en ejecutarse debido a la sobrecarga de las llamadas a funciones, pues debido a cada llamada recursiva, se asigna algo de memoria en la pila para almacenar parámetros y variables locales.

# Cuestiones II-D

## Cuestión 1

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

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

**Gráfico, Gráfico de líneas

Descripción generada automáticamente**

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Por la forma de la gráfica, los tiempos parecen ajustarse a una función rectilínea creciente, del orden de O(N) .

Si analizamos el código de la función examinada, podemos observar que realmente usamos dos bucles. El bucle más interno se hace a través del método min\_heapify(), el cual no recorre la lista entera, sino que lo hace dando "saltos" entre lo que serían los nodos padres e hijos del heap, es decir, el coste de la función heapify es O(lg(N)) .

Por otro lado de la función de creación del heap contiene el bucle externo dónde dicho bucle realizará O(N) llamadas. Y es que este bucle se encarga de recorrer todos los nodos internos del árbol, desde el último padre hasta la raíz. Realmente no se recorren todos los elementos del array, pero técnicamente se está recorriendo gran parte del array elemento a elemento, así que se considera que tiene un coste de O(N)

De esta manera, en conjunción, se obtiene un coste total de O(N∗lg(N))

Sin embargo, como vemos en las gráficas, los resultados no se asemejan tanto a esta función, acercándose en mayor medida a la función O(N) . Aunque el límite superior seleccionado en el caso anterior es correcto, este puede ser más estrecho.

Si nos fijamos bien en el desarrollo del algoritmo, podemos ver que efectivamente la función de heapify sigue una complejidad temporal de O(prof(h)) ; sin embargo si especificamos en mayor medida, podemos observar cómo dicho árbol aumenta en cada ejecución del bucle externo comenzando en un árbol de profundidad 1 (comenzamos directamente en el padre del último nodo), donde el coste claramente es O(1) al tener como máximo un único cambio que realizar en el min\_heapify(); hasta el de la profundidad completa (desde la raíz), dentro del cual el coste es O(lg(N)) , pues podría realizar tantos cambios como profundidad tenga el árbol.

Esto lleva a la conclusión de que heapificar nos lleva un tiempo distinto en cada iteración de la creación del heap siguiendo la serie: 1∗N/4+2∗N/8+3∗N/16+...+prof(h)∗1 (multiplicamos la cantidad máxima de cambios ( k ) por el número de padres de la capa ( ceil(N/2k+1) ) en cada interacción).

A partir de la suma obtenemos el sumatorio: sum(k∗ceil(N/2k+1)) .

Si cogemos el caso general, quitamos el techo del segundo factor y podemos sacar factor común N/2 , de modo que el sumatorio queda N/2∗sum(k/2k) .

Ahora trabajamos con el sumatorio, lo llamaremos s . Si lo extendemos, s=1/2+2/4+3/8+4/16+... . Ahora 2s=2/2+4/4+6/8+8/16+...=1+2/2+3/4+4/8+... . Si los restamos reordenando los términos según su denominador, s=2s−s=1+(2/2−1/2)+(3/4−2/4)+(4/8−3/8)+...=1+1/2+1/4+1/8+... , la cual es la serie geométrica (suma de un número infinito de términos que tiene una razón constante entre sus términos sucesivos siguiendo una progresión descendente) lo cual converge a la suma (s/(1−r))=(1/(1−(1/2)))=(1/(1/2))=2 .

De este modo, el resultado final del coste es N/2∗2=N

## Cuestión 2

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

Suponemos tamaño del array N .

En la función select\_min\_heap() primero se crea el max heap de tamaño k , que supone O(k) como ya vimos en el apartado anterior; seguido de un bucle de N−k iteraciones, pues los primeros k elementos se usan para la creación del heap. Dentro del bucle, se coge la raíz para realizar comprobaciones, este coste será de O(lg(N)) en el peor de los casos.

Se obtiene finalmente una complejidad temporal equivalente a O((N−k)∗lg(k))+O(k)=O((N−k)∗lg(k))

## Cuestión 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 obtendrán estos k elementos.**

Al finalizar la ejecución del select\_min\_heap(), dentro del max heap que se crea, obtenemos los k elementos menores, estando colocado en la raiz el elemento mayor dentro de los k menores, es decir, el que ocuparía la posición k-estima.

Por otro lado, en la otra implementación (select\_min\_heap\_2()) se van haciendo extracciones (sacando en cada iteración el elemento mayor, o menor si empleamos un min heap) hasta llegar al que ocupe la posición k, es decir bastaría con guardar esos valores por ejemplo en una lista y en esta ocasión el resultado si que serían los k elementos ordenados.

## Cuestión 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? ¿Como?**

Siguiendo la filosofía del select\_min\_heap, se podría realizar con dos bucles (uno empleado en la creación del heap) y otro para la selección sin embargo no sería un bucle doble si no dos bucles separados con incluso mejor rendimiento que un único bucle que recorra toda la array ya que tendría complejidad (como vimos en el apartado anterior) O(2)+O((N−2)∗lg(2))=O(N−2)+O(2)=O(N−2) lo cual supone un rendimiento superior a O(N) y especialmente a O(N2)

[FINAL DE DOCUMENTO]