PRACTICA DIVIDE Y VENCERÁS

AÑO ACADÉMICO: 2020-2021

ASIGNATURA: AED2

ALUMNOS: David Fernández Expósito y

Pablo Tadeo Romero Orlowska

DNI: [redacted]

GRUPO: PCEO

SUBGRUPO DE PRÁCTICAS: 1.4 (PCEO)

PROFESOR: NORBERTO MARÍN PÉREZ

PROBLEMA: 2

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN 3

2. PSEUDOCÓDIGO Y EXPLICACIÓN DEL ALGORITMO 3

2.1 TIPO SOLUCIÓN 3
2.2 MÉTODO PEQUEÑO 3
2.3 MÉTODO DIVIDE_VENCERÁS 4
2.4 MÉTODO SOLUCIÓN_DIRECTA 5
2.5 MÉTODO COMBINAR 6

3. ESTUDIO TEÓRICO DEL TIEMPO DE EJECUCIÓN 9

3.1 SOLUCIÓN DIRECTA 9
3.2 COMBINAR 11
3.3 PEQUEÑO 12
3.4 DIVIDE_VENCERÁS 12

4. PROGRAMACIÓN DEL ALGORITMO 15

5. VALIDACIÓN DEL ALGORITMO 19

6. ESTUDIO EXPERIMENTAL DEL TIEMPO DE EJECUCIÓN 22

7. CONTRASTE ESTUDIO TEÓRICO Y EXPERIMENTAL 31

8. CONCLUSIONES, VALORACIONES PERSONALES Y PROBLEMAS ENCONTRADOS 36

9. REFERENCIAS BIBLIOGRÁFICAS 36

1. INTRODUCCIÓN

En este trabajo se utilizará la técnica divide y vencerás para resolver el problema propuesto por el profesor, en este caso el número 2. En particular, se usará esta técnica dividiendo el problema en dos subproblemas. El ejercicio asignado tiene el siguiente enunciado:

2) Dadas dos cadenas A y B de la misma longitud n, y un natural m ≤n, hay que encontrar el índice p de inicio de la subcadena de tamaño m con más diferencia total en valor absoluto entre los caracteres en cada posición de A y B (suma de los m valores |A[i]-B[i]| entre las posiciones p y p+m-1). Devolver el índice p de comienzo de la solución y el valor de la mayor diferencia total. En caso de empate, será válida cualquiera de las soluciones óptimas.

En cuanto a los archivos entregados en el .rar, se puede encontrar esta memoria en pdf, así como los archivos de código. El archivo llamado "divideVenceras.cpp" es el código principal, y los archivos "generador.cpp" y "generador.hpp" se corresponden con el generador. También se incluyen varios archivos Excel donde se podrán encontrar las tablas que dan lugar a las gráficas mostradas en este documento, y un pdf con algún ejemplo significativo.

2. PSEUDOCÓDIGO Y EXPLICACIÓN DEL ALGORITMO

2.1 TIPO SOLUCIÓN

```
Tipo Solución{
      posición, suma: enteros;
}
```

Se decidió usar un tipo *solución* pues es mejor devolver esto, que contiene la posición y la suma (que es lo que se pide), en lugar de devolver solamente la posición y cada vez que se quiera usar la solución obtenida de algún subproblema tener que hacer un bucle para calcular la suma.

2.2 MÉTODO PEQUEÑO

```
Pequeño (inicio, fin: enteros){
      si (fin – inicio + 1 < 2m || fin–inicio+1<n/1000) entonces
            devolver verdadero;
      sino
            devolver falso;
```

```
}
```

Este método simplemente devuelve *true* cuando el problema es lo suficientemente pequeño como para usar la *solución directa*. Se decidió usar la condición n < 2m. pues si n fuese menor que m, no se podría llamar a *solución directa*, y si n ≥ 2m aún se puede descomponer en otros dos subproblemas donde aplicar *solución directa*.

Un tema que queremos comentar es por qué tomamos que, si n es muy grande y la m muy pequeña, también se ejecuta *solución directa* si tamaño < n/1000, pues podría ser poco eficiente dividir tantas veces dado el caso. Aunque es cierto que esto provocaría que siempre se hiciese el mismo número de divisiones, se añade esta otra opción para los casos en los que no merece la pena dividir tantas veces, cuando n es muy grande y m muy pequeño. En teoría, lo idóneo hubiese sido determinar el valor de pequeño teóricamente, pero en este caso podríamos llegar a la conclusión de que lo mejor es no dividir nunca, pues es mejor hacer *solución directa* simplemente, y claramente este no es el objetivo de esta práctica. Por este motivo se decidió poner esta opción en *pequeño* con un valor razonable, para, en la práctica, ganar algo de eficiencia.

2.3 MÉTODO DIVIDE_VENCERÁS

```
Divide_Vencerás (inicio, fin: enteros){
      si (pequeño(inicio, fin)) entonces
            devolver Solución_Directa(inicio, fin);
      sino
            sol1 := Divide_Vencerás (inicio, (inicio + fin)/2);
            si (sol1.suma = sol_optima) entonces
                  devolver sol1;
            finsi
            sol2 := Divide_Vencerás ((inicio + fin)/2 + 1, fin);
            si (sol2.suma = sol_optima) entonces
                  devolver sol2;
            finsi
            devolver Combinar (sol1, sol2, (inicio + fin)/2);
      finsi;
}
```

Como se observa, para conseguir una mayor eficiencia en el caso de que se encuentra la solución óptima, se evita llamar a la función combinar o a la siguiente llamada recursiva en su caso, ya que, al haber obtenido la solución óptima, ya sabemos que no se puede encontrar una solución mejor y no tiene ningún sentido seguir la ejecución.

2.4 MÉTODO SOLUCIÓN_DIRECTA

```
Solucion_Directa (inicio, fin: enteros){
      i:= inicio;
      j:= inicio + m - 1;
      pos := 0;
      suma := 0;
      suma_max := 0;
      para h := i…j hacer
            suma := suma + diferencia[h];
      finpara
      i := i + 1;
      j := j + 1;
      suma_max := suma;
      mientras (j <= fin y suma_max < m*25) hacer
            suma = suma + diferencia[j] – diferencia[i-1];
            si (suma > suma_max) entonces
                  suma_max := suma;
                  pos := i;
            finsi
            i := i + 1;
            j := j + 1;
      finmientras
      Solucion s;
      s.posicion := pos;
      s.suma := suma;
      devolver s;
```

```
}
```

Como ocurría en el método *divide y vencerás,* en caso de en algún momento encontrar la solución óptima, no se sigue con la ejecución del método pues ya sabemos que no se puede encontrar una solución mejor.

Otro tema importante a comentar es cómo hemos intentado optimizar este método. La primera manera de implementarlo que se nos ocurrió era simplemente ir iterando desde el comienzo del array con subarrays de tamaño m, y en cada iteración sumar los m valores para obtener la suma correspondiente. Sin embargo, esto nos llevaría a un orden cuadrático que se puede evitar con la solución planteada. En este caso, se reutiliza la suma anterior, para en cada iteración sumar el valor de la "nueva" posición añadida, y restar el valor de la posición que se ha "salido" del subarray. Esta suma es la que en cada iteración se compara con la mayor suma hasta el momento. Así, se consigue una mayor eficiencia.

2.5 MÉTODO COMBINAR

```
Combinar (sol1, sol2: Solución; punto_medio: entero){
      Solucion sol3;
      sol3.suma := 0;
      sol3.posicion := 0;
      suma := 0;
      si ((sol1.posicion + sol1.posicion + m)/2 >= punto_medio –
m + 2) entonces
            suma := sol1.suma;
            para h:=sol1.posicion…punto_medio – m + 1 hacer
                  suma := suma – diferencia[h];
            finpara
            para h:=sol1.posicion + m…punto_medio + 1 hacer
                  suma := suma + diferencia[h];
            finpara
            sol3.suma := suma;
            sol3.posicion := punto_medio – m + 2;
            i := punto_medio – m + 3;
            j := punto_medio + 2;
```

```
            mientras(j <= punto_medio+ m – 1 y sol3.suma < 25*m){
                  suma = suma - diferencia[i-1] + diferencia[j];
                  si (suma > s.suma){
                        sol3.suma := suma;
                        sol3.posicion := i;
                  finsi
                  i := i + 1;
                  j := j + 1;
            finmientras;


      sino si ((sol2.posicion+ sol2.posicion + m)/2 <=
punto_medio+m-1) entonces
            suma := sol2.suma;
            para h:= punto_medio+m…sol2.posicion + m - 1 hacer
                  suma := suma – diferencia[h];
            finpara
            para h:=sol2.posicion - 1…punto_medio hacer
                  suma := suma + diferencia[h];
            finpara
            sol3.suma := suma;
            sol3.posicion := punto_medio;
            i := punto_medio + m - 2;
            j := punto_medio - 1;
            mientras(j <= punto_medio- m + 2 y sol3.suma < 25*m){
                  suma := suma – diferencia[i+1] + diferencia[j];
                  si (suma >= sol3.suma){
                        sol3.suma := suma;
                        sol3.posicion := j;
                  finsi
                  i := i - 1;
                  j := j - 1;
```

```
            finmientras;
        sino
           sol3:=Solucion_Directa(punto_medio–m+2,punto_medio+m-1);
        finsi
        si (sol1.suma >= sol3.suma) entonces
            sol3 = sol1;
        finsi
        si (sol2.suma > sol3.suma) entonces
            sol3 = sol2;
        finsi
        devolver sol3;
}
```

Se puede observar fácilmente que este método es el más complicado de implementar. Esto se debe a cómo intentamos mejorar su eficiencia. En un primer lugar, pensamos en simplemente llamar a *solución directa* en este método (pasando como parámetros las posiciones (punto_medio–m+2) y (punto_medio+m-1)) para obtener la solución de "en medio", y luego comparar con las dos soluciones obtenidas en sendas llamadas recursivas. Sin embargo, de nuevo, vimos que se podía reutilizar la suma de las soluciones ya calculadas como se hizo en el método anterior. Por esta razón, se comprueba si alguna de las dos soluciones está lo suficientemente cerca al punto medio para que pueda ser "reutilizada". Tomamos que una solución está lo suficientemente cerca si su punto medio está en el intervalo que se trata en la función *combinar*, pues es cuando se observa una reducción en número de instrucciones realizadas. Si es así, se usa el valor de su suma para sumarle y restarle los valores correspondientes y poder obtener así la primera suma del bucle de una manera más eficiente.

Nótese que, si no se puede reutilizar ninguna de las dos soluciones que se pasan como parámetro, se llama a *solución directa*, aunque esto se comentó en clase que no era lo ideal. En nuestro caso se consideró que sí que era correcto pues al final se debe hacer lo mismo, por lo que en lugar de "copiar" código se decidió hacer esta llamada a *solución directa*. Otra cosa a comentar es que, si se reutiliza una de las dos soluciones, no se reutiliza la otra. Esto lo implementamos así porque tras reflexionar llegamos a la conclusión de que no supone ninguna ventaja usar ambas soluciones, es mejor usar una para calcular la primera suma y luego ir reutilizando las sumas que se hacen para obtener la siguientes.

3. ESTUDIO TEÓRICO DEL TIEMPO DE EJECUCIÓN

En este apartado se hará un estudio teórico del tiempo de ejecución a partir del planteamiento en pseudocódigo. Para ello, se dividirá el estudio en los distintos métodos.

Como se puede observar en el pseudocódigo, se trabajará todo el rato con el array diferencias, que almacena las diferencias en valor absoluto entre las dos cadenas.

3.1 SOLUCIÓN DIRECTA

Empezaremos por el mejor caso. Para la *solución directa*, como planteamos el algoritmo de manera que si se encuentra la solución óptima (m*25) se devuelve directamente, está claro que el mejor caso que podemos encontrar sería que apareciese esta solución óptima en la primera suma calculada. De esta manera, el tiempo para el mejor caso quedaría:

$$g_m(n,m) = 5 + 2m + 3 + 1 + 3 = 12 + 2m$$

Nótese que el "+1" de la primera expresión es debido a la comprobación del bucle, al que no se entra pues hemos supuesto que se encuentra la solución óptima a la primera.

Por tanto, vemos que $g(n,m) \in \Omega(m)$

Si nos centramos ahora en el peor caso, lo primero que se nos ocurrió es que este caso sería cuando tanto el bucle como el condicional dentro de él se realizan el máximo de veces posible. Por tanto, si se reflexiona se llega a la conclusión de que, si el array diferencias estuviese ordenado de menor a mayor, se tendría que las sumas serían cada vez mayores y se entraría en el condicional en cada iteración del bucle. Como a solución directa se llamará cuando n < 2m, lo único que debemos pensar es que el peor caso es cuando el trozo del array que se está tratando está ordenado de menor a mayor en cuanto a las sumas de m posiciones seguidas se refiere. Nótese que no tenemos que preocuparnos en este caso por el hecho de que 25 es la diferencia máxima posible y el array no podría ser creciente indefinidamente, pues a solución directa solo se entra si n < 2m y en este caso no encontramos este problema.

Un ejemplo de un array diferencias que cumpliría lo explicado sería uno con los siguientes valores:

0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9 ….

En este ejemplo, la m = 3.

Así, en este peor caso tendremos que:

$$g_M(n,m) = 5 + 2m + 3 + (n-m)(7) + 1 + 4 = 13 + 7n - 5m$$

Podríamos entonces concluir que: $g(n,m) \in O(n-m)$.

En cuanto al tiempo promedio, éste no hemos podido obtenerlo con exactitud, pero sí que hemos podido razonar sobre él e intentar aproximarnos lo más posible.

Primero, razonamos cuáles son las probabilidades que hay de que se ejecute el condicional. Para que se entre en este condicional, es necesario que la suma que se ha calculado en la correspondiente iteración del bucle sea la mayor de todas. En particular se necesita que sea mayor que la anterior. La suma anterior y la actual se diferencian en un solo elemento, por lo que la probabilidad de que la suma sea mayor que la anterior se reduce a la probabilidad de que sea mayor el elemento de la suma en el que difieren. Así, esta probabilidad es ½. Si queremos que sea mayor a la suma anterior de la anterior, necesitamos que la suma de sus dos últimos números sea mayor a la de los dos primeros de la anterior de la anterior, y como hay ½ de probabilidades de que el primer número de estos sea mayor, tendremos que la probabilidad de que esta suma sea mayor es ¼ (por la probabilidad condicionada, ½ * ½). Así, si nos centramos en las instrucciones del condicional, tendremos:

$$\sum_{i=1}^{n-m} \frac{1}{2^i} * 2$$

Nótese que el límite del sumatorio es n - m, pues son las veces que se ejecuta el bucle, siempre que no se obtenga la solución óptima. Sabemos que esto no es correcto del todo, pues la suma tiene un valor máximo, pero podríamos intentar acercarnos al tiempo promedio teniendo en cuenta que los valores de las sumas solo pueden ser desde 0 hasta 25*m. Por lo tanto, podríamos añadir al conteo de instrucciones la probabilidad de que se encuentre la solución óptima, que sería de 1/(25*m + 1). Podríamos llegar a una expresión del estilo:

$$g_p(n,m) = 5 + 2m + 3 + 4 + 1 + \sum_{i=1}^{n-m} \left(\frac{25 * m}{25 * m \ + \ 1} * \left(5 + \frac{1}{2^i} * 2\right)\right)$$

Nótese que como la probabilidad de que la suma sea la óptima es de 1/(25*m + 1), debemos poner en la expresión la probabilidad de que no ocurra esto, que es cuando se ejecutaría el bucle. Por lo tanto, aparece el $\frac{25*m}{25*m+1}$. Claro está, en caso de que se cumpla que la solución es la óptima, se debería salir del bucle, pero esto no sabemos cómo representarlo en la fórmula obtenida. Además, tampoco es del todo correcta la expresión de la probabilidad de que la suma sea la óptima, pues realmente las sumas anteriores afectan a la probabilidad de que la siguiente suma sea o no la óptima, y esto cambiaría la probabilidad (si en la suma anterior es no participa ningún

25, es imposible que en la siguiente suma aparezca la suma óptima). De hecho, seguramente la probabilidad de que la suma sea la óptima es aún menor que la que hemos tenido en cuenta en la expresión obtenida.

Entonces, como conclusión, observamos que no tenemos orden exacto para la función *solución directa* pues no coinciden el orden en el peor caso y en el mejor caso. Lo único que podemos afirmar es que el tiempo en el caso promedio estará entre estos dos.

3.2 COMBINAR

En *combinar*, hay que notar que hay tres opciones: que se reutilice la solución de "la parte izquierda", que se reutilice la de la "parte derecha" o que se tenga que llamar a *solución directa* porque no se pueda reutilizar ninguna solución ya calculada. Empezamos por ver qué ocurre en este último caso.

Si se tiene que llamar a *solución directa* (aplicada a la subcadena que queda "en medio"), el peor y mejor caso se corresponden con el peor y mejor caso de *solución directa*. Tendremos así:

$$f_m(n,m) = 4 + 1 + 1 + g_m(2m-2,m) + 3 = 9 + 12 + 2m = 21 + 2m$$

$$f_M(n,m) = 4 + 1 + 1 + g_M(2m-2,m) + 5 = 11 + 13 + 7(2m-2) - 5m = 10 + 9m$$

Nótese que en una se suman al final 3 instrucciones y en otra se suman 5, pues en el mejor caso no se entra en los condicionales del final, mientras que en el peor sí.

Ahora vemos qué ocurre en los otros dos casos. Hay que notar que, en cualquiera de los dos casos, el número de instrucciones será el mismo pues se hace exactamente lo mismo, pero de derecha a izquierda en lugar de al revés.

En el mejor caso:

$$f_m(n,m) = 1 + 2 + 2 + 4 + 1 + 3 = 13$$

En el mejor caso, suponemos que la solución que se reutiliza está "lo mas pegada" posible al punto medio, por lo que solo hay que restar un número y sumar otro para obtener la primera suma. Además, si esta suma es la óptima, no se entra en el bucle y en ningún condicional del final, por lo que obtenemos este orden constante.

En el peor caso:

$$f_M(n,m) = 1 + 2 * \frac{m}{2} + 2 * \frac{m}{2} + 4 + (m-2)(6) + 1 + 5 = 11 + 2m + 6m - 12 = 8m - 1$$

En el peor caso, la solución que se reutiliza "está lo más alejada posible" del punto medio y el bucle se ejecuta el máximo número de veces entrando siempre en el condicional, además de entrar en todos los condicionales del final.

Un detalle a comentar es que, si la solución que se reutiliza es la de la "parte derecha", en la expresión mostrada deberíamos añadir una instrucción más correspondiente con la comprobación del condicional correspondiente.

Entonces, observamos que, en *combinar*, el mejor caso es cuando se puede reutilizar alguna solución para calcular la suma y esta es además la solución óptima. En el peor caso, obtenemos expresiones que nos llevan al mismo orden de complejidad tanto si llamamos a *solución directa* y nos encontramos con el peor caso, como si reutilizamos alguna solución y nos encontramos en el peor caso. Así:

$$f(n,m) \in O(m) \qquad\qquad f(n,m) \in \Omega(1)$$

Vemos cómo no tenemos orden exacto para la función *combinar* pues no coinciden ambos órdenes.

3.3 PEQUEÑO

La función *pequeño* simplemente comprueba si el tamaño del problema es más pequeño que 2m. Por lo tanto, es claramente de orden exacto $\Theta(1)$.

3.4 DIVIDE_VENCERÁS

Como ya se comentó en *pequeño,* si m fuese suficientemente pequeña, se usaría como condición que el tamaño sea menor que n/1000. Sin embargo, el estudio se realizará usando n < 2m, es decir, en función de m, pues realmente podríamos reducir el otro caso a uno particular de este. La m es variable, pero si en particular tomase el valor n /2000, la condición que obtenemos es la misma. Por lo tanto, el esquema general sería el siguiente:

$$t(n,m) = g(n,m) \qquad si\ n < 2 * m$$

$$t(n,m) = 2 * t(\frac{n}{2}, m) + f(m) \quad si\ n \geq 2 * m$$

Ahora, si razonamos en el mejor caso, este será cuando se encuentre la solución óptima a la primera. Entonces, no se llegará a realizar la segunda llamada recursiva (se entrará en el condicional que hay justo tras terminar la llamada recursiva) ni la llamada a *combinar*,

simplemente se resolverá la primera llamada recursiva que terminará cuando se encuentre la solución óptima a la primera cuando se llame a solución directa. Tendremos entonces:

$$t_m(n,m) = t_m(\frac{n}{2},m) + 2 = t_m(\frac{n}{4},m) + 4 = t_m(\frac{n}{8},m) + 6 = \ldots$$
$$= t_m(\frac{n}{2^{(\log_2 n - \log_2 2m)}},m) + 2 * (\log_2 n - \log_2 2m)$$

Es importante destacar que en el caso del log de n y el log de 2m se toma la parte entera inferior. Este valor es el número de divisiones que hay que hacer hasta que n < 2*m. Sin embargo, si la m fuese potencia de 2, habría que sumar 1 a esta expresión, pero tomamos la expresión usada sin que esto suponga pérdida de generalidad. Por lo tanto, cuando se llegue a ese valor, se llamará a solución directa, por lo que se queda (usando que $2^{\log_2 n - \log_2 2m} = \frac{n}{2m}$):

$$t_m(n,m) = g_m\left(\frac{n}{2^{(\log_2 n - \log_2 2m)}},m\right) + 2 * (\log_2 n - \log_2 2\,m)$$
$$= 12 + 2m + 2 * (\log_2 n - \log_2 2m)$$

Por lo tanto, observamos que $t(n,m) \in \Omega(m + \log n - \log m)$. Esto tiene sentido, pues, aunque en solución directa es cierto que, para el mejor caso, no influye la n, a la hora de ir haciendo las llamadas recursivas es claro que la n tendrá influencia.

Para el peor caso, será cuando nos encontremos con el peor caso en *solución directa* y en *combinar*. Sin embargo, hay que comentar ciertos detalles sobre el peor caso del algoritmo. El peor caso lo encontraremos cuando nunca se encuentre la solución óptima, para que en solución directa nunca se "pare" de buscar una solución mejor. Por lo tanto, el array debería estar ordenador de menor a mayor de manera que las sumas de m posiciones seguidas vayan aumentando de uno en uno. Sin embargo, bien es sabido que como mucho la diferencia entre dos caracteres será 25, por lo que el array deberá ser como el explicado para el peor caso de *solución directa*, pero cuando se llegue a (ejemplo con m=3) …24 25 25, a partir de este momento, se volverá a comenzar desde 0 de la misma forma, y así sucesivamente. Otro detalle a comentar es que, si bien este array supone el peor caso de *solución directa*, en algunas ejecuciones de *solución directa* no se entrará en el condicional dentro del bucle, pero esto no supone un impacto en el tiempo pues lo importante es que el bucle se realice entero. En cuanto a combinar, se tomará el peor caso, que ya vimos que daba igual cuál de las tres opciones tuviésemos pues en el peor caso daban lugar al mismo orden.

Así en ese caso, tendríamos una expresión como la siguiente:

$$t_M(n,m) = 2 * t_M\left(\frac{n}{2}, m\right) + f_M(m) = 2^2 t_M\left(\frac{n}{4}, m\right) + 2f_M(m) + f_M(m) = \cdots$$

$$= 2^{\log_2 n - \log_2 2m} t_M\left(\frac{n}{2^{\log_2 n - \log_2 2m}}, m\right) + \sum_{j=0}^{\log_2 n - \log_2 2m} 2^j f_M(m)$$

Es importante destacar que en el caso del logaritmo de n y de 2m se toma la parte entera inferior. Este valor es el número de divisiones que hay que hacer hasta que n < 2*m (aunque si m es potencia de 2, habría que sumar 1, como ya se comentó). Por lo tanto, cuando se llegue a ese valor, se llamará a *solución directa*, por lo que se queda:

$$t_M(n,m) = 2^{\log_2 n - \log_2 2m} \times c \times \left(\frac{n}{2^{\log_2 n - \log_2 2m}} - m\right) + \sum_{j=0}^{\log_2 n - \log_2 2m} 2^j \times d \times m$$

Operando, y teniendo en cuenta que $2^{\log_2 n - \log_2 2m} = \frac{n}{2m}$, obtenemos:

$$t_M(n,m) = \frac{n}{2m} \times cm + dm\left(\frac{1 - \frac{n}{m}}{-1}\right) = n\left(\frac{c}{2} + d\right) - dm$$

$$t(n,m) \in O(n-m)$$

Una cuestión que debemos aclarar es que en las expresiones que se han usado que contenían los logaritmos, esos logaritmos los hemos usado tal cual, pero en realidad la expresión tenía en cuenta la parte superior para el logaritmo de m y la parte inferior para el de n. Se decidió obviar este detalle a la hora de realizar los cálculos para agilizarlos y hacerlos más sencillos. Por ejemplo, cuando se hace la observación de que $2^{\log_2 n - \log_2 m} = \frac{n}{m}$, si no hubiésemos obviado las partes superiores e inferiores no hubiésemos podido simplificar de esta manera los cálculos.

En cuanto al tiempo promedio, dado que no hemos sido capaces de obtenerlo para *solución directa*, no podemos obtenerlo para el método *divide y vencerás*, el método general. Sabemos que el tiempo promedio se encontrará entre el peor y el mejor, no necesariamente en el medio justo. Sin embargo, por cómo es la expresión "aproximada" que se obtuvo y por cuál es el número de veces que se puede llegar a ejecutar el sumatorio de dicha expresión (se realiza n – m veces), podríamos deducir que este caso estará más cercano al orden del peor caso que del mejor, pues la probabilidad de que aparezca la solución óptima y se salga del bucle son muy reducidas.

```
int n;
int m;
int * diferencias;

struct Solucion{
    int posicion;
    int suma;
};

Solucion solucion_directa (int inicio, int fin){
    int i = inicio;
    int j = inicio + m-1;
    int suma = 0;
    int suma_max = 0;
    int pos = i;
    //Se calcula la primera suma
    for (int h = i; h <= j; h++){
        suma = suma + diferencias[h];
    }
    suma_max = suma;
    i++;
    j++;
    while (j <= fin && suma_max < m*25) {
        //Se hace la suma usando la anterior
        suma = suma + diferencias[j] - diferencias[i - 1];
        //Actualizamos la mejor solución si procede.
        if (suma > suma_max){
            suma_max = suma;
            pos = i;
        }
        i++;
        j++;
    }
    Solucion sol;
```

```
    sol.suma = suma_max;
    sol.posicion = pos;
    return sol;
}


Solucion combinar (Solucion sol1, Solucion sol2, int punto_medio){
    Solucion s;
    s.suma = 0;
    s.posicion = 0;
    int suma = 0;
    //Si la solucion de la izquierda esta lo suficientemente cerca se
reutiliza
    if ((sol1.posicion + sol1.posicion + m)/2 >= punto_medio - m + 2){
        suma = sol1.suma;
        //Se calcula la primera suma usando la solución de la izquierda
        for (int h = sol1.posicion; h <= punto_medio - m + 1; h++){
            suma = suma - diferencias[h];
        }
        for (int h = sol1.posicion + m; h <= punto_medio + 1; h++){
            suma = suma + diferencias[h];
        }
        s.suma = suma;
        s.posicion = punto_medio - m + 2;
        int i = s.posicion + 1;
        int j = punto_medio + 2;
        //Iteramos haciendo las demás sumas
        while (j <= punto_medio + m - 1 && s.suma < 25*m){
            suma = suma + diferencias[j] - diferencias[i - 1];
            if (suma > s.suma){
                s.suma = suma;
                s.posicion = i;
            }
            i++;
            j++;
        }
```

```
        //Si la solucion de la derecha esta lo suficientemente cerca se
reutiliza
    } else if ((sol2.posicion + sol2.posicion + m)/2 <= punto_medio + m - 1){
        suma = sol2.suma;
        //Se calcula la primera suma usando la solución de la derecha
        for (int h = punto_medio + m; h <= sol2.posicion + m - 1; h++){
            suma = suma - diferencias[h];
        }
        for (int h = sol2.posicion - 1; h >= punto_medio; h--){
            suma = suma + diferencias[h];
        }
        s.suma = suma;
        s.posicion = punto_medio;
        int i = punto_medio + m - 2;
        int j = punto_medio - 1;
        //Iteramos haciendo las demás sumas
        while (j >= punto_medio - m + 2 && s.suma < 25*m){
            suma = suma + diferencias[j] - diferencias[i + 1];
            if (suma >= s.suma){
                s.suma = suma;
                s.posicion = j;
            }
            i--;
            j--;
        }
    //Si no se puede usar ninguna de las dos soluciones se llama a solcuión
directa
    } else{
        s = solucion_directa(punto_medio - m + 2, punto_medio + m - 1);
    }
    //Se devuelve la mayor de las tres soluciones, la más a la izquierda en
caso de empate.
    if (sol1.suma >= s.suma){
        s = sol1;
    }
    if (sol2.suma > s.suma){
        s = sol2;
```

```
    }
    return s;
}

bool pequeno(int inicio, int fin){
    if (fin - inicio + 1 < 2*m || fin - inicio + 1 < n/1000){
        return true;
    }
    return false;

}

Solucion divide_venceras(int inicio, int fin){
    if (pequeno(inicio, fin)){
        return solucion_directa(inicio, fin);
    } else{
        Solucion sol1 = divide_venceras(inicio, (inicio + fin)/2);
        //Si es la solución optima se devuelve directamente.
        if (sol1.suma == m*25){
            return sol1;
        }
        Solucion sol2 = divide_venceras((inicio + fin)/2 + 1, fin);
        //Si es la solución optima se devuelve directamente.
        if (sol2.suma == m*25){
            return sol2;
        }
        return combinar(sol1, sol2, (inicio + fin)/2);
    }
}
```

No se muestra el código del main, pues no queremos que se extienda excesivamente esta sección de código y en este caso es un método principal muy sencillo sin nada especial. No obstante, se puede encontrar en los archivos de código entregados junto a esta memoria.

Tampoco se muestra el código de los generadores, el cuál se muestra en el apartado 6 de la presente memoria y también se puede encontrar en los ficheros de código entregados.

En cuanto al uso de variables globales, vemos cómo se usan 3. Usamos dos variables globales de tipo entero para almacenar el valor de n y m, y un puntero a entero (cuya memoria se reserva en los generadores, donde también se inicializa el array al que apunta, y se libera en el main). Usamos un puntero para poder tener un array diferencias de tamaño distinto según el valor de n. El motivo de usar estas tres variables como globales fue la comodidad y conveniencia de no tener que ir pasando por parámetro en cada función estas variables.

5. VALIDACIÓN DEL ALGORITMO

Para validar el correcto funcionamiento del algoritmo implementado, se probaron múltiples casos promedios, extremos, generados aleatoriamente, de distintas longitudes y distintos valores de m, etc. y se solucionaron haciendo uso tanto de la técnica divide y vencerás como de la solución directa. Así, fuimos capaces de comprobar que, efectivamente, de ambas maneras se obtiene el mismo resultado.

Para que esta forma de validar el algoritmo, que usa la técnica divide y vencerás, tenga sentido, es claro que primero teníamos que estar seguros del correcto funcionamiento de la solución directa, lo cual fue lo primero que se hizo nada más comenzar la realización de esta práctica. El hecho de que este sea un algoritmo relativamente sencillo permite que sea fácil verificar su corrección a mano. Por ejemplo, se hizo que en cada ejecución se mostrase por pantalla tanto las cadenas como el array diferencias, para poder confirmar que la solución era correcta. Claramente esto último no lo pudimos hacer con tamaños excesivamente grandes.

Por lo tanto, lo que se hizo fue hacer una copia del fichero .cpp y modificar el main para que se ejecutase el número de veces que se quiera (se pregunta al usuario por este valor, así como si quiere el mejor, peor o caso promedio, y por el valor de n y m). En este bucle, se van generando distintas cadenas, llamando a divide y vencerás y solución directa en cada iteración, y sólo en el caso en que la posición o la suma devuelta por la solución directa difiera con la devuelta con la solución obtenida usando divide y vencerás se muestra por pantalla el array diferencias y los valores de las dos soluciones. Claro está, para la validar el algoritmo, el objetivo es que no se muestre ninguna discrepancia por pantalla. A continuación, se muestra el código del método main modificado para hacer estas pruebas:

```
int main (){

    int n_casos = 0;

    cout << "Numero de casos" << endl;

    cin >> n_casos;

    cout << "Inserte el tamaño del array" << endl;

    cin >> n;

    cout << "Inserte la m que desee" << endl;

    cin >> m;

    if (m > n){

        cout << "La m debe ser menor o igual que la n" << endl;

        return 0;

    }

    cout << "Inserte 1 si desea el peor caso, 2 si desea el mejor, 3 si desea un caso
promedio" << endl;

    int caso;

    cin >> caso;

       for (int i = 0; i < n_casos; i++){

              cout << i << endl;

               if (caso == 1){

                   diferencias = genera_peor_caso(n,m);

               } else if (caso == 2){

                   diferencias = genera_mejor_caso(n,m);

               } else if (caso == 3){

                   diferencias = genera_caso_promedio(n,m);

               } else {

                   cout << "Opción inválida" << endl;

                   return 0;

               }

               Solucion sol = divide_venceras(0, n-1);

               Solucion sol2 = solucion_directa(0, n -1);

               if (sol.posicion != sol2.posicion){

                   cout << "POSICION DISTINTA! " << endl;

                   //IMPRESIÓN ARRAY DIFERENCIAS PARA COMPROBACIÓN

                   for (int j = 0; j < n; j++){

                       cout << diferencias[j] << " ";

                   }

                   cout << endl;
```

```
                    cout << "Solucion directa posicion:suma " << sol2.posicion
<<":"<<sol2.suma << endl;

                    cout << "Divide y venceras posicion:suma " << sol.posicion << ":" <<
sol.suma << endl;

                }

                if (sol.suma != sol2.suma){

                    cout << "SUMA DISTINTA! " << endl;

                    //IMPRESIÓN ARRAY DIFERENCIAS PARA COMPROBACIÓN DE QUE TODO OK

                    for (int j = 0; j < n; j++){

                        cout << diferencias[j] << " ";

                    }

                    cout << endl;

                    cout << "Solucion directa posicion:suma " << sol2.posicion
<<":"<<sol2.suma << endl;

                    cout << "Divide y venceras posicion:suma " << sol.posicion << ":" <<
sol.suma << endl;

                }

                delete[] diferencias;

            }

        }
```

Al llevar a cabo la validación, se hicieron 1000 casos del peor caso, del mejor y del caso promedio (haciendo uso de los generadores) para distintos valores de n y de m (1000 casos para cada par de valores), excepto si la n era muy grande, donde se decidió hacer 200 o 100 casos pues con 1000 el tiempo que consumía el programa era demasiado alto.

El código de los distintos generadores de casos de prueba se explica y muestra en el punto siguiente de esta memoria. Estos generadores sirvieron para el posterior estudio experimental del tiempo de ejecución, así como para la validación del algoritmo. Además, con en el fichero .rar entregado, se puede encontrar otro fichero pdf con algunos ejemplos significativos de casos de prueba usados. Como los vamos a mostrar, claro está que aparecen casos con n pequeña para que sea fácilmente legible.

6. ESTUDIO EXPERIMENTAL DEL TIEMPO DE EJECUCIÓN

Para realizar el estudio experimental del tiempo de ejecución, como se pedía en la especificación de la práctica, se implementaron generadores de casos de prueba. Se implementaron tres generadores:

- Generador de caso promedio: este generador simplemente, dadas la n y la m, genera dos cadenas aleatorias de tamaño n. A partir de estas dos cadenas, se construye el array diferencias.
- Generador de mejor caso: este generador, dadas la n y la m, genera dos subcadenas de manera que la primera tiene m letras 'a' seguidas al principio, y la segunda m letras 'z' seguidas al principio, para tener la solución óptima a la primera en el array diferencias. Cabe destacar que el resto de las cadenas se decidió no hacerlo aleatorio, pues esto supone un coste innecesario, pues da igual que el resto de la cadena sea aleatorio o cualquier otra cosa "fijada", ya que no se hará uso de esa parte por encontrarse justo a la primera la solución óptima. Por tanto, se decidió poner en ambos casos la letra 'p' hasta llegar a la longitud n de ambos arrays. Podría haberse puesto, claro está, cualquier otra cosa.
- Generador peor caso: en este generador se crea una cadena con todo letras 'a'. La segunda cadena será del estilo "aaabbbccc…" si por ejemplo m=3, es decir, tendrá grupos de m letras en orden ascendente. Esto será si n < 26*m. Si no, la cadena repetirá el mismo patrón anterior de nuevo hasta el final de la cadena. En un principio pensamos en concatenar "…yzzz…" sucesivamente, pero de esta manera en los problemas de *solución directa* de la parte derecha sólo se entrará al condicional una vez. Por lo tanto, con la solución planteada sabemos que se entrará más veces al condicional, manteniendo el mismo número de iteraciones del bucle (el máximo), pues nunca se debe encontrar la solución óptima.

A continuación, se muestra el código en C++ de estos tres generadores de casos de prueba:

```
int * genera_mejor_caso(int n, int m){
    string cadena1 = "";
    string cadena2 = "";
    for (int i = 0; i < m; i++){
        cadena1 = cadena1 + "a";
        cadena2 = cadena2 + "z";
    }
```

```
    for (int i = m; i < n; i++){
        cadena1 = cadena1 + "p";
        cadena2 = cadena2 + "p";
    }
    int * dife = new int[n];
    for (int i = 0; i < n; i++){
        dife[i] = abs(cadena1[i] - cadena2[i]);
    }
    return dife;
}

int * genera_peor_caso(int n, int m){
    string cadena1 = "";
    string cadena2 = "";
    int i = 0;
    //creamos una cadena con todo 'a'
    for (int h = 0; h < n; h++){
        cadena2 = cadena2 + "a";
    }
    //la otra cadena la hacemos como se explica en la memoria.
    while(cadena1.length() < n){
        char letra = 'a' + i;
        int j = 0;
        if (i < 25){
            while (cadena1.length() < n && j < m){
                cadena1 = cadena1 + letra;
                j++;
            }
        } else {
            while (cadena1.length() < n && j < m-1){
                cadena1 = cadena1 + letra;
                j++;
            }
        }
```

```
        if (i < 25){
            i++;
        } else {
            i = 0;
        }
    }
    int * dife = new int[n];
    for (int i = 0; i < n; i++){
        dife[i] = abs(cadena1[i] - cadena2[i]);
    }
    return dife;
}

//En el caso promedio se generan cadenas aleatorias.
int * genera_caso_promedio (int n, int m){
    string cadena1 = "";
    string cadena2 = "";
    srand(time(NULL) + rand());
    int random;
    for (int i = 0; i < n; i++){
        random = rand()%26;
        char letra1 = 'a' + random;
        cadena1 = cadena1 + letra1;
        random = rand()%26;
        char letra2 = 'a' + random;
        cadena2 = cadena2 + letra2;
    }
    int * dife = new int[n];
    for (int i = 0; i < n; i++){
        dife[i] = abs(cadena1[i] - cadena2[i]);
    }
    return dife;
}
```

Para la medición del tiempo era imprescindible ser capaces de medirlo sin contar todo lo que supone crear las cadenas, crear el array diferencias, etc. Por lo tanto, tras investigar un poco por internet, decidimos hacer uso de la biblioteca "chrono" para medir el tiempo transcurrido entre que se llama al método *divide y vencerás* y se sale de él. Este es el tiempo que nos interesa y el que se mostraba por pantalla durante el estudio experimental de nuestro programa. El código del main utilizado quedó así:

```
int main (){
    cout << "Inserte el tamaño del array" << endl;
    cin >> n;
    cout << "Inserte la m que desee" << endl;
    cin >> m;
    if (m > n){
        cout << "La m debe ser menor o igual que la n" << endl;
        return 0;
    }
    cout << "Inserte 1 si desea el peor caso, 2 si desea el mejor, 3 si desea
un caso promedio" << endl;
    int caso;
    cin >> caso;
    if (caso == 1){
        diferencias = genera_peor_caso(n,m);
    } else if (caso == 2){
        diferencias = genera_mejor_caso(n,m);
    } else if (caso == 3){
        diferencias = genera_caso_promedio(n,m);
    } else {
        cout << "Opción inválida" << endl;
        return 0;
    }
    //Medimos el tiempo antes de llamar a divide y vencerás
    auto t0 = std::chrono::high_resolution_clock::now();
    Solucion sol = divide_venceras(0, n-1);
    //Medimos e tiempo después
    auto t1 = std::chrono::high_resolution_clock::now();
    //calculamos la diferencia
```

```
    auto time = std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0);

    //Mostramos el tiempo

    cout << "Execution time: " << time.count()*1e-9 << endl;

    //Hacemos el problema también con solución directa directamente para ver que
sale lo mismo

    Solucion sol2 = solucion_directa(0, n -1);

    cout << "Posicion " << sol.posicion << " y diferencia maxima " << sol.suma
<< endl;

    cout << "sol directa es (pos:suma) " << sol2.posicion << ":" << sol2.suma
<< endl;

    delete[] diferencias;

}
```

Para llevar a cabo el estudio experimental se hicieron 20 ejecuciones del peor caso, del mejor y del promedio para distintos valores de m fijos. Para cada valor fijado de m se hicieron pruebas con distintos valores de n. Por lo tanto, se hicieron 20 ejecuciones para n = 1000 y m = 250, n = 5000 y m = 250, n = 10000 y m = 250 y así. Se probó para m = 250, m = 500 y m = 1000. En cuanto a n, se probó con n = 1000, n = 5000, n = 10000, n = 25000, n=50000, n = 75000, n = 100000, n = 150000, n = 200000 y n = 250000. A partir de este último valor de n ya se empieza a notar que el ordenador se ralentiza y hubiese supuesto mucho tiempo tomar medidas para valores más grandes.

Para tomar los datos que se usarán en los gráficos, simplemente se hizo la media de los valores obtenidos en las 20 ejecuciones. Decidimos hacer 20 ejecuciones por un tema de tiempo, para que nos diese tiempo ha tomar todos los datos. Otra opción que sopesamos fue, como se hizo para validar el algoritmo, usar un bucle que hiciese muchas ejecuciones y calculase la media de tiempos. Sin embargo, pensamos que, al tratarse de una sola ejecución del ejecutable, podría no representar una media del tiempo en distintos estados de la máquina. Por esta razón, se decidió hacer 20 ejecuciones, lanzada a mano cada una, para así tener en cuenta que el procesador, por ejemplo, puede no tener la misma carga en cada ejecución.

Así, a continuación, se muestran las gráficas obtenidas para los distintos valores de m para el mejor caso. En cada gráfica se puede ver en rojo la línea de tendencia logarítmica (ponemos esta pues es lo que supuestamente tendría que salir según el estudio teórico, pero esta comparación ya la haremos en el punto siguiente de la memoria), así como el valor de R^2, que indica cómo se ajusta dicha línea de tendencia a los datos, siendo mejor cuanto más cercano a 1.

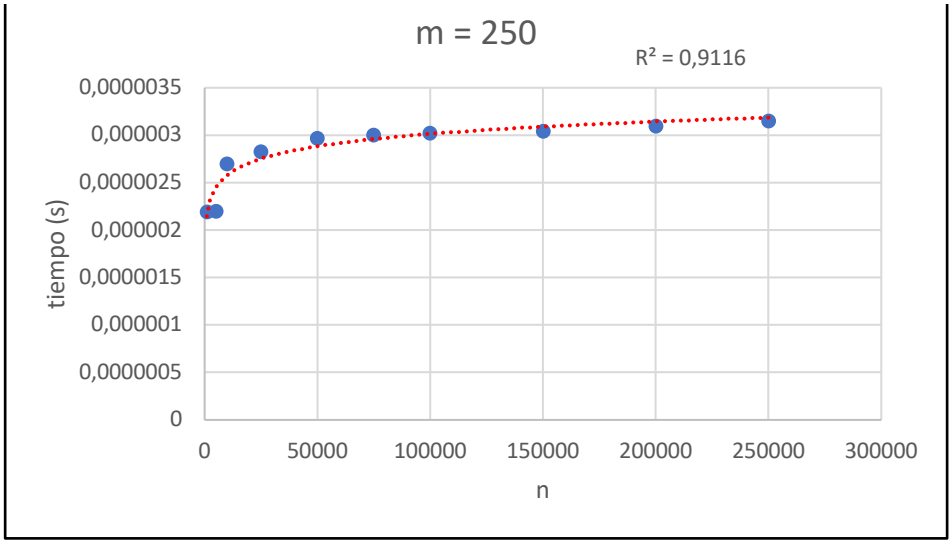

Figura 1. Gráfica para los tiempos obtenidos en el mejor caso para m = 250. Fuente: propia.

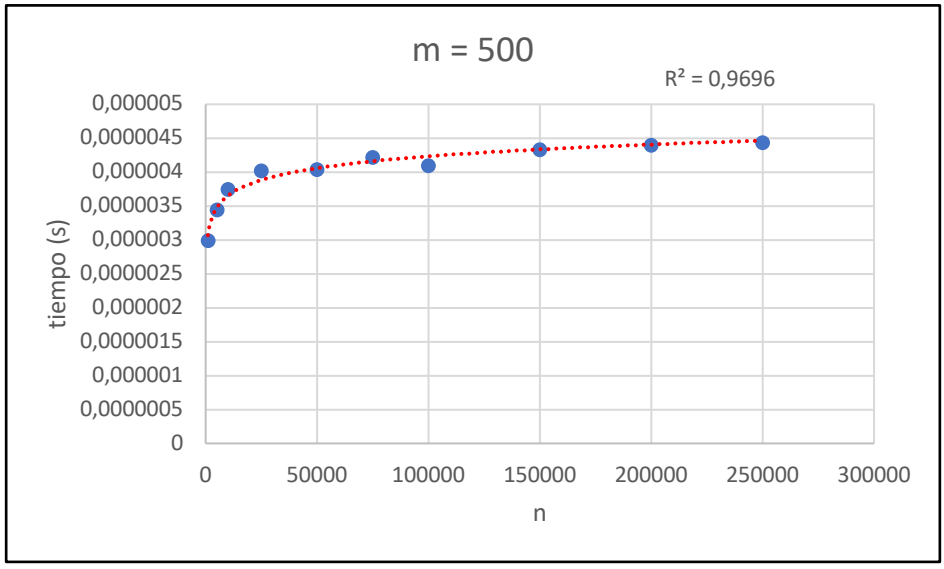

Figura 2. Gráfica para los tiempos obtenidos en el mejor caso para m = 500. Fuente: propia.

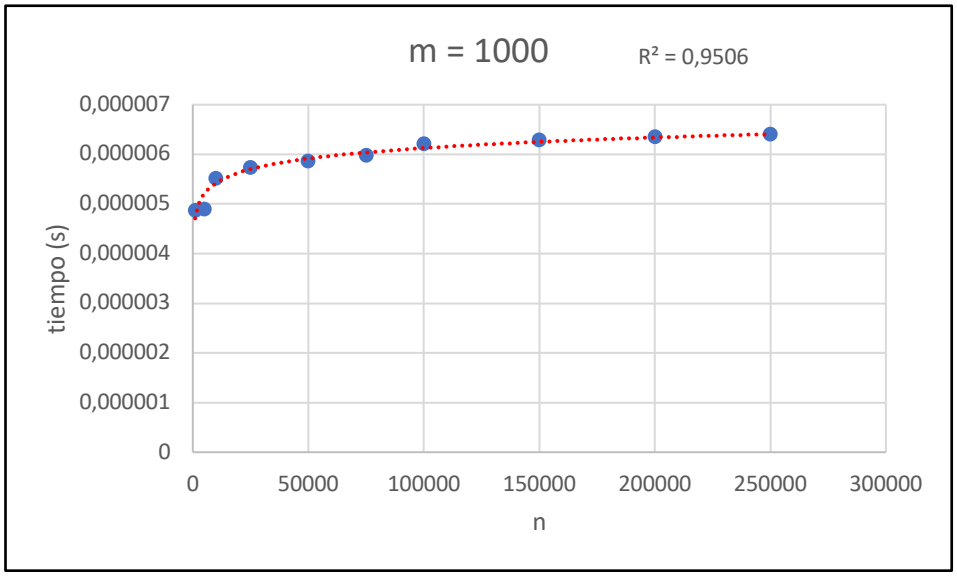

Figura 3. Gráfica para los tiempos obtenidos en el mejor caso para m = 1000. Fuente: propia.

Se puede observar que los datos siguen un crecimiento logarítmico según n, pues la línea de tendencia se acopla bastante bien. Por otro lado, se observa cómo conforme aumentamos la m, para el mismo valor de n, el tiempo aumenta, por lo que el valor de m también tiene influencia en el tiempo de ejecución del mejor caso.

A continuación, se muestran los datos para el caso promedio para los distintos casos de m. El formato seguido es idéntico al caso anterior:

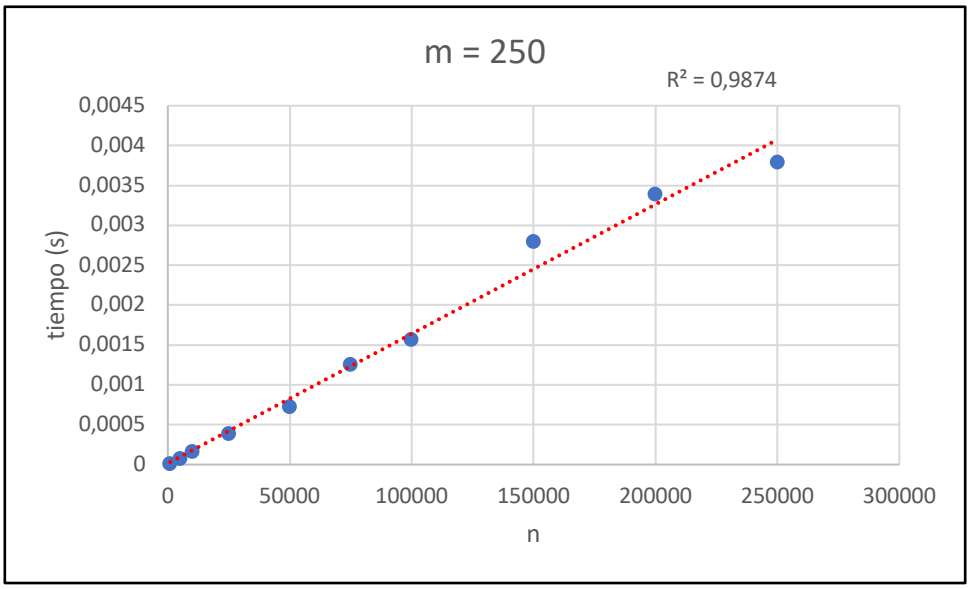

Figura 4. Gráfica para los tiempos obtenidos en el caso promedio para m = 250. Fuente: propia.

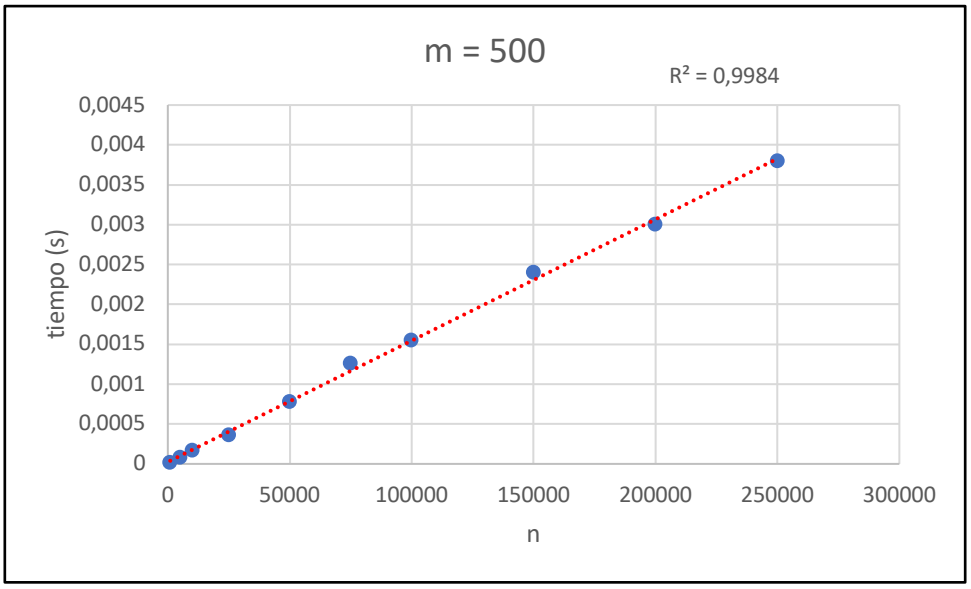

Figura 5. Gráfica para los tiempos obtenidos en el caso promedio para m = 500. Fuente: propia.

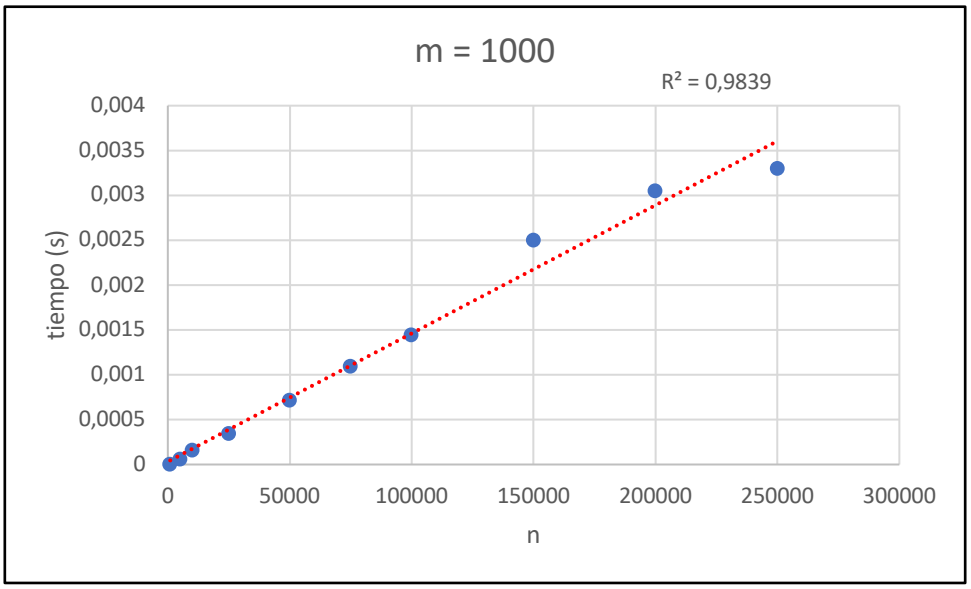

Figura 6. Gráfica para los tiempos obtenidos en el caso promedio para m = 1000. Fuente: propia.

Observamos cómo los tiempos en el caso promedio parecen crecer de manera lineal. En este caso, la línea de tendencia es una línea de tendencia lineal, pues era la que mejor se acoplaba según se puede observar en el valor de R^2. Además, los tiempos parecen decrecer conforme aumentamos m para un mismo valor de n, por lo que se podría pensar que el tiempo promedio es de orden (n – log(m)) o incluso (n – m).

Por último, se muestran las gráficas obtenidas para los distintos valores de m para el peor caso. El formato será el mismo que en los casos anteriores, aunque ahora la línea de tendencia que se mostrará será de tipo lineal pues esto es lo que salía en el estudio teórico (esta comparación la realizaremos en el siguiente apartado):

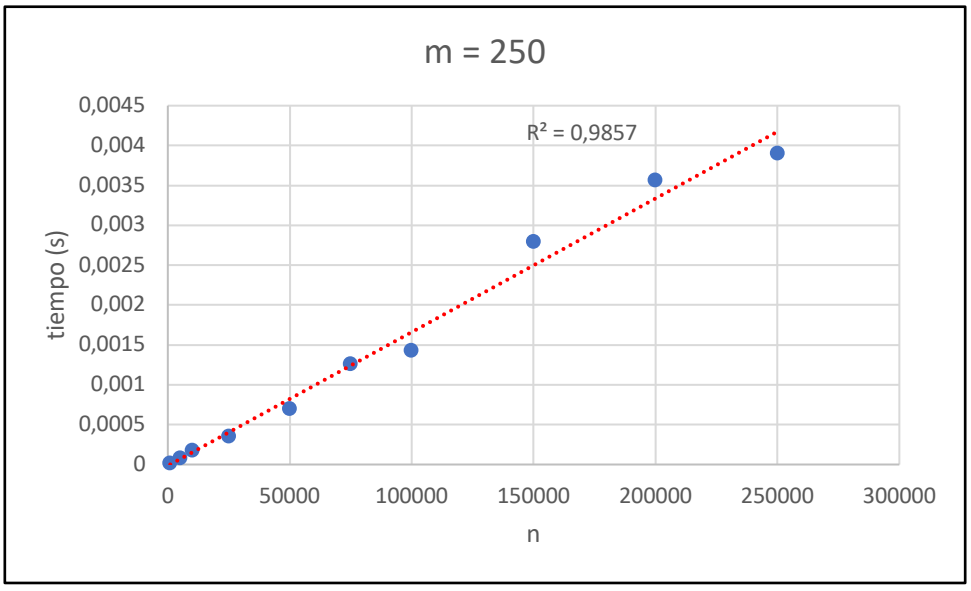

Figura 7. Gráfica para los tiempos obtenidos en el caso peor para m = 250. Fuente: propia.

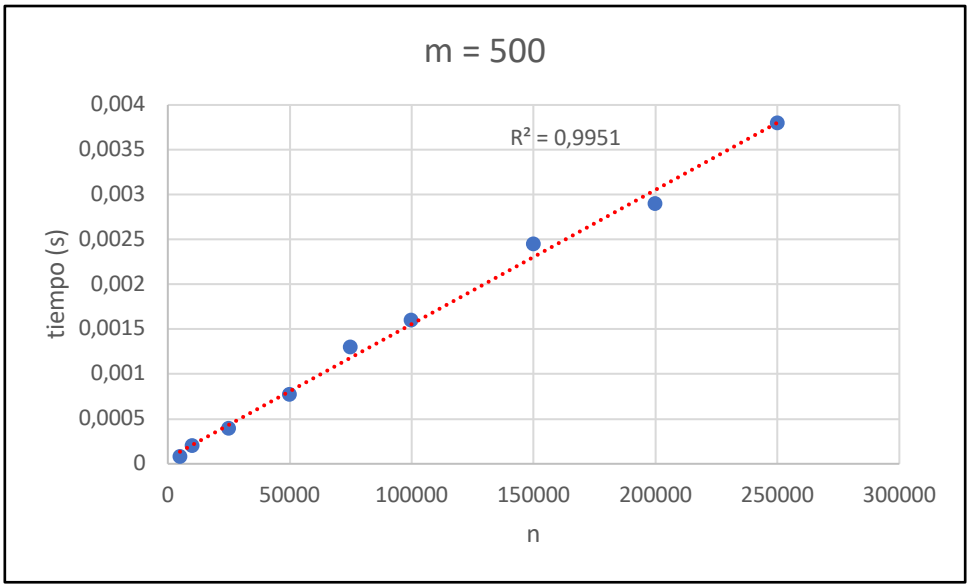

Figura 8. Gráfica para los tiempos obtenidos en el caso peor para m = 500. Fuente: propia.

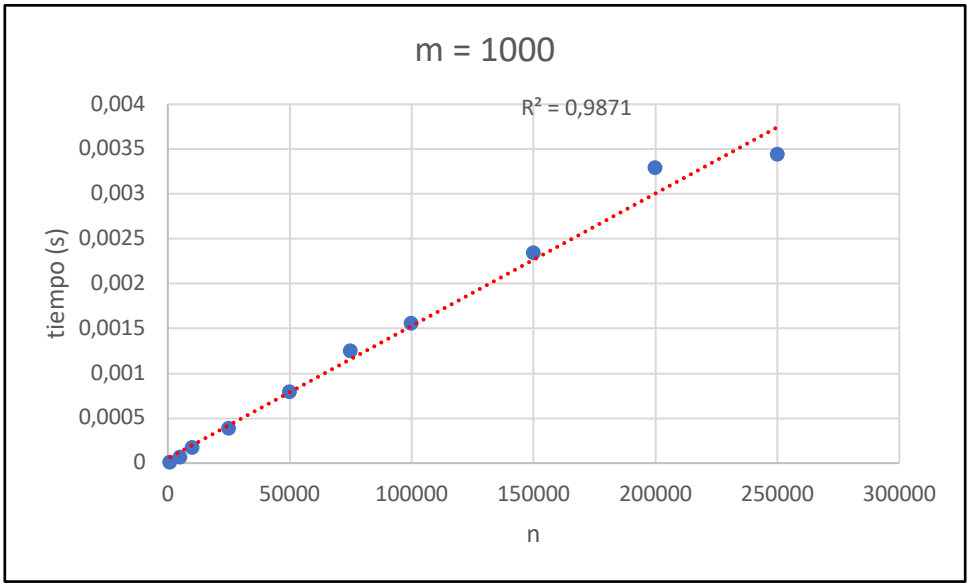

Figura 9. Gráfica para los tiempos obtenidos en el caso peor para m = 1000. Fuente: propia.

Podemos observar como, de nuevo, parece que los tiempos crecen de forma lineal. Esto quiere decir que, si nos fijamos también en los datos obtenidos para el caso promedio, ambos crecen de forma similar. De hecho, los tiempos no son muy distintos, incluso en algunos casos el caso promedio ha llegado promediar un tiempo mayor que el caso peor. Esto se puede deber a ciertas lagunas que no podemos controlar a la hora de plantear el peor caso. No obstante, de esto podemos deducir que el caso promedio se encuentra más cercano al peor caso que al mejor, lo cual tiene sentido pues las posibilidades de que se obtenga la solución óptima son muy bajas para

un m suficientemente grande, por lo que los bucles se realizarán casi el mismo número de veces en el caso promedio y en el peor caso. Además, vemos cómo los tiempos tienden a ser ligeramente inferiores conforme se aumenta la m.

7. CONTRASTE ESTUDIO TEÓRICO Y EXPERIMENTAL

En este apartado haremos una comparativa entre los resultados obtenidos en el estudio experimental y en el estudio teórico del tiempo de ejecución. Para ello, primero mostraremos una serie de gráficas que serán de gran ayuda. Se mostrarán dos tipos de gráficas: el primer tipo mostrará los datos obtenidos en el estudio experimental para un mismo valor de m, y para el peor caso, el mejor y el promedio. El otro tipo mostrará, para uno de los tres casos (mejor, peor, promedio) los valores obtenidos para cada valor de m.

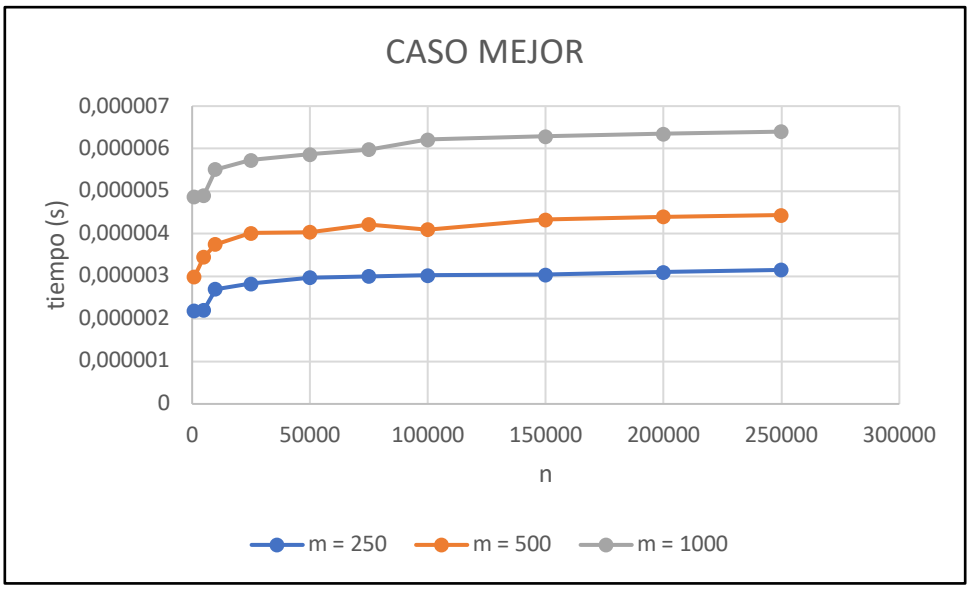

Figura 10. Gráfica para los tiempos obtenidos en el mejor caso. Fuente: propia.

Si nos fijamos en esta gráfica y en las del apartado anterior se puede observar que, según el estudio experimental, fijada una m, los tiempos de ejecución en el mejor caso parecen seguir un crecimiento logarítmico. Volviendo al estudio teórico realizado en el punto 4 de esta memoria, vemos que esto concuerda con lo obtenido teóricamente, pues se calculó que el algoritmo cumplía que $t(n,m) \in \Omega(m + \log n - \log m)$. Esto quiere decir que podremos encontrar, fijada una m, un múltiplo de la función $m + \log n - \log m$ que acotaría inferiormente los tiempos de ejecución de nuestro algoritmo. Además, también se puede observar en esta gráfica (y en las del estudio experimental), que conforme se aumenta el valor de m, aumenta el tiempo

de ejecución, lo cual también concuerda con el orden calculado para el algoritmo en el mejor caso.

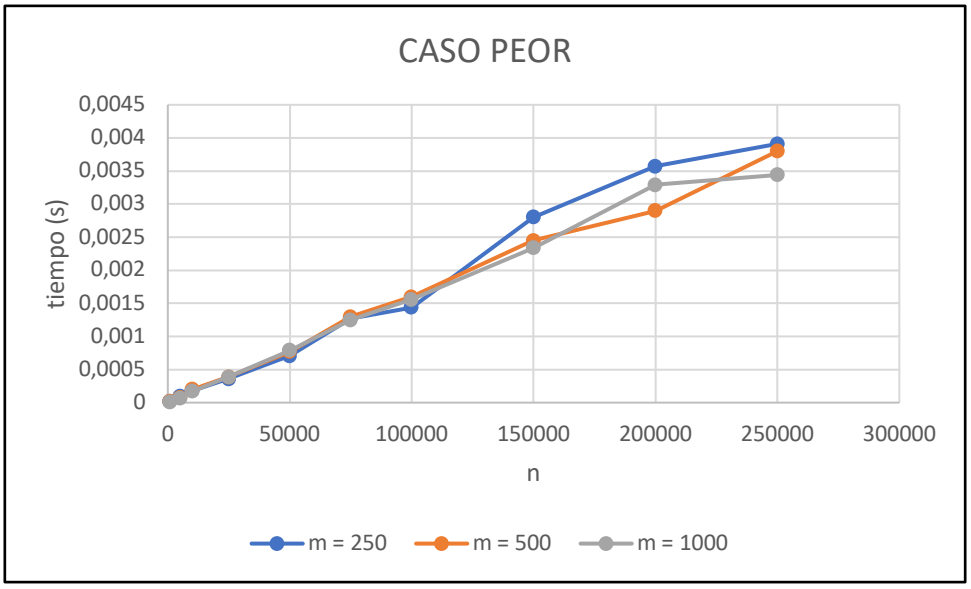

Figura 11. Gráfica para los tiempos obtenidos en el peor caso. Fuente: propia.

En esta gráfica y en las presentes en el anterior estudio experimental podemos observar cómo el tiempo de ejecución en el peor caso crece de forma lineal conforme se aumenta el valor de n. Si recordamos lo que se obtuvo en el estudio teórico, vemos cómo efectivamente esto tiene sentido, pues obtuvimos que $t(n, m) \in O(n - m)$. Por lo tanto, fijado un valor de m, podríamos encontrar un múltiplo de $n - m$ que nos serviría para acotar superiormente los tiempos de ejecución de nuestro algoritmo. Además, aunque no se hace muy evidente en el estudio experimental, sí que se puede notar cómo, aunque ligeramente, conforme se aumenta el valor de m disminuye el tiempo de ejecución, lo cual también refleja el orden obtenido de forma teórica, si bien es cierto esperábamos que este descenso fuese un poco más notorio.

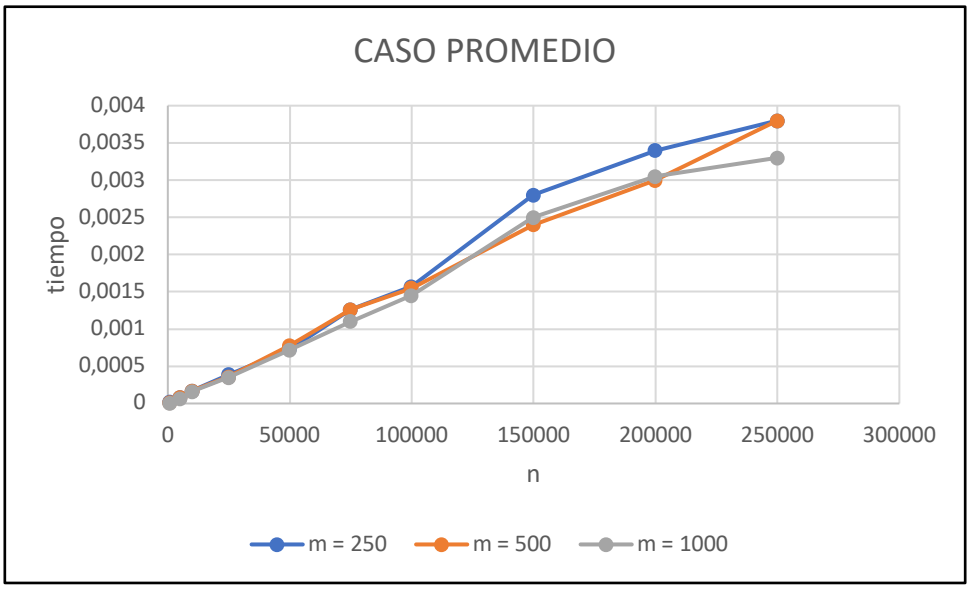

Figura 12. Gráfica para los tiempos obtenidos en el caso promedio. Fuente: propia.

Vemos cómo, según el estudio experimental anterior y la presente gráfica, el tiempo en el caso promedio parece crecer de manera lineal y que, a mayores valores de m, el tiempo disminuye ligeramente. Además, si nos fijamos en el punto anterior, los valores que se obtienen en las gráficas del tiempo del caso promedio son muy parecidos, e incluso en alguna ocasión muy ligeramente superiores, a los obtenidos en el peor caso. Esto parece indicar que el tiempo promedio esta más cerca del peor caso que del mejor, lo cual concuerda si pensamos en cuál es el mejor caso. En el caso promedio, las probabilidades de obtener la solución óptima son muy reducidas con un m suficientemente grande, por lo que el número de iteraciones de los bucles principales será muy similar, o el mismo, en el caso promedio y en el peor caso. Esto se puede ver fácilmente también en las siguientes gráficas que se muestran, que para cada valor de m muestra los datos obtenidos en cada uno de los tres casos:

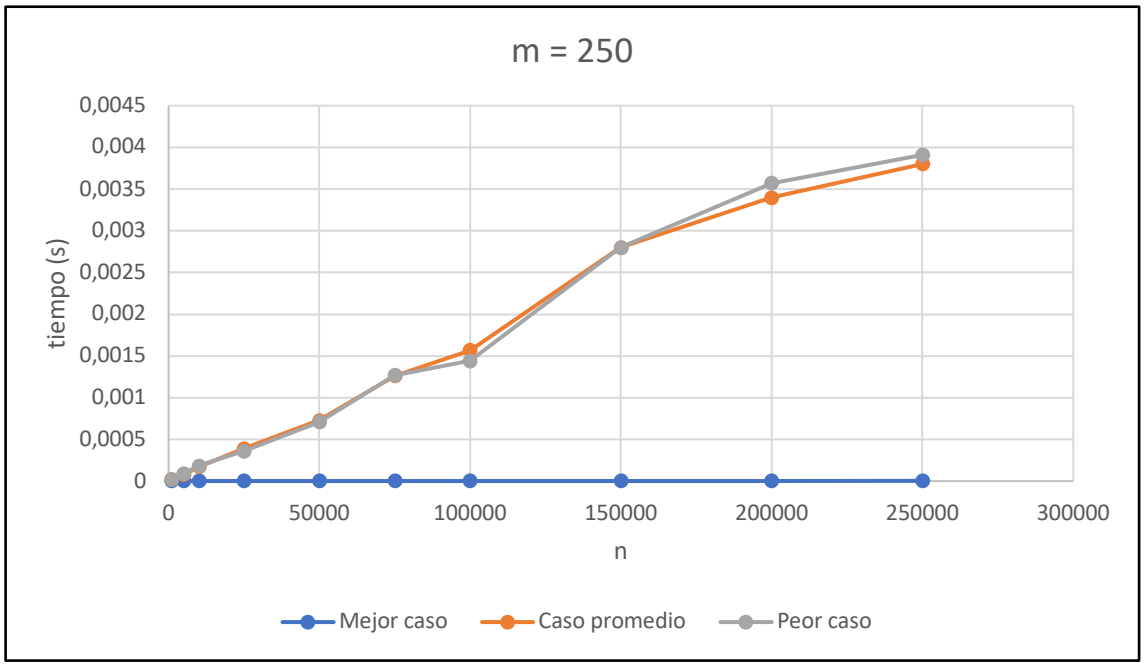

Figura 13. Gráfica para los tiempos obtenidos m = 250 en los tres casos. Fuente: propia.

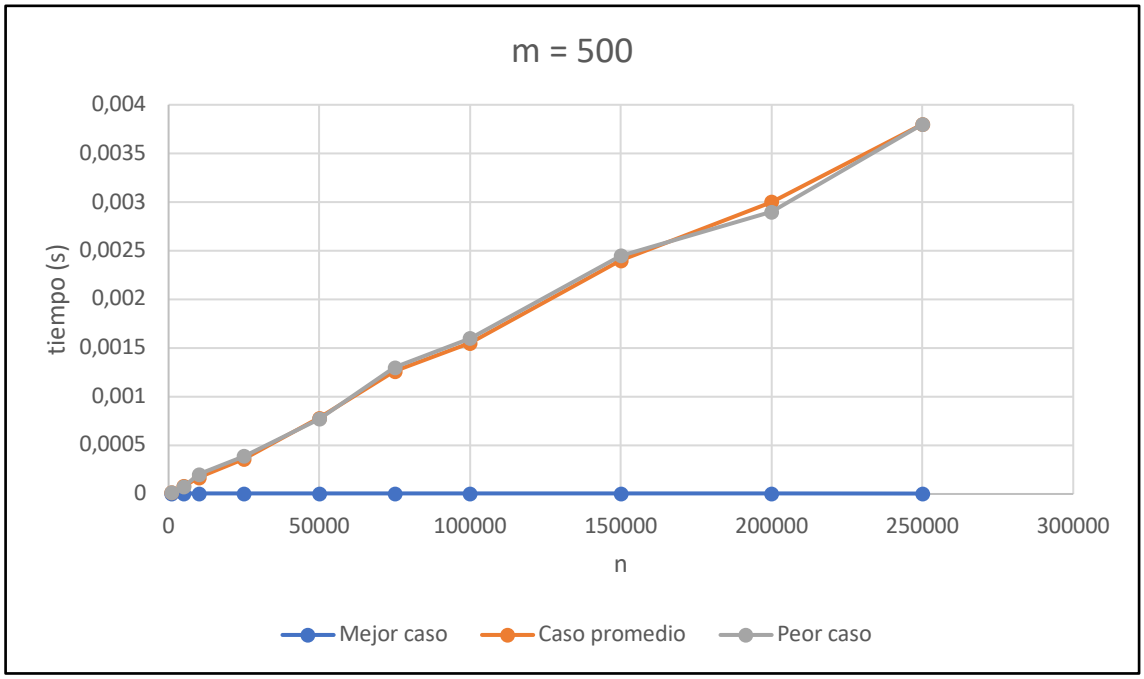

Figura 14. Gráfica para los tiempos obtenidos m = 500 en los tres casos. Fuente: propia.

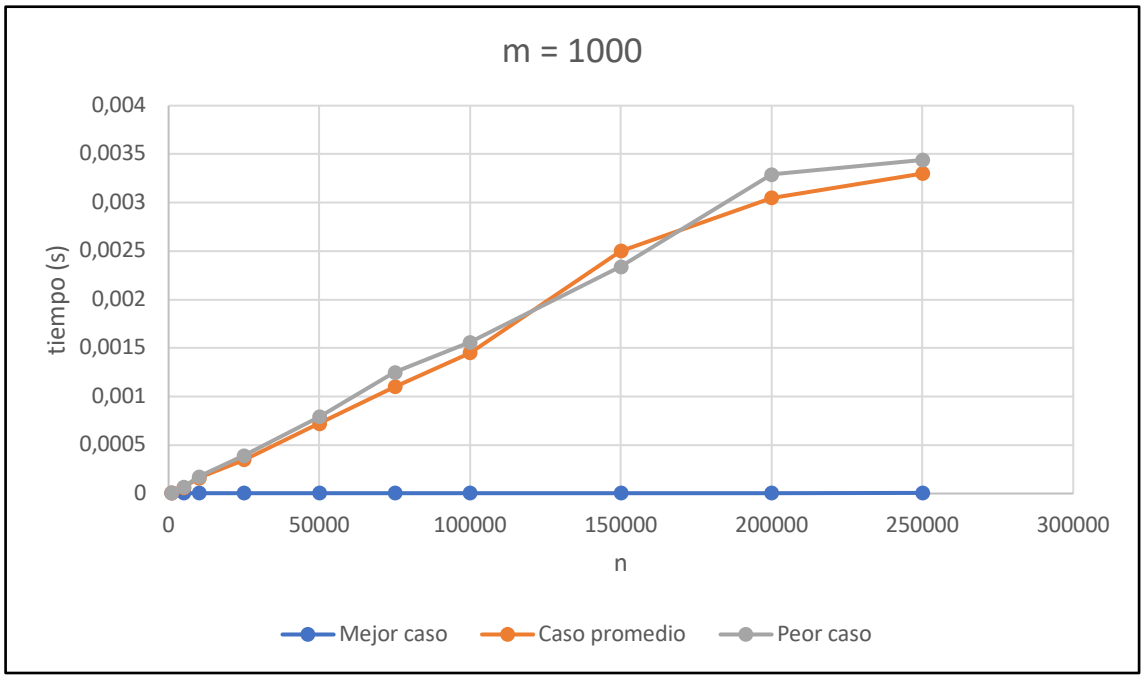

Figura 15. Gráfica para los tiempos obtenidos m = 1000 en los tres casos. Fuente: propia.

En estas tres últimas gráficas se puede observar cómo el mejor caso supone un tiempo muy pequeño si lo comparamos con el promedio y el peor caso. De hecho, no se puede ni observar el crecimiento logarítmico que se obtiene en el mejor caso, lo cual sí pudimos ver en las otras gráficas mostradas tanto al comienzo de este apartado como en el estudio experimental de esta memoria, debido a la escala que se utilizaba. Además, como se comentó con anterioridad, vemos cómo las gráficas del peor caso y el caso promedio van muy juntas todo el rato, normalmente con el peor caso ligeramente por encima, aunque en algún punto se llega a poner por encima el caso promedio. Esto vuelve a mostrar lo que ya se comentó, que debido a que el caso óptimo es tan poco probable de obtener, y que haría que los bucles principales en el caso promedio se ejecutasen menos veces, podemos deducir que el caso promedio estará muy próximo al peor caso. Si volvemos al estudio teórico realizado, a pesar de no ser capaces de estudiar teóricamente el tiempo promedio, sí que llegamos a una expresión que, por cómo era el sumatorio que tenía (se realizaría n – m veces), podía hacernos pensar que, efectivamente, el caso promedio estaría más cerca del orden del peor caso que del mejor, como finalmente hemos concluido. En cuanto a la razón por la que a veces el caso promedio supera en tiempo al peor, creemos que debe a cómo hemos planteado el peor y la naturaleza de nuestro problema, pues como la diferencia máxima entre dos caracteres es 25, no podemos tener un array creciente infinitamente, lo cual limita “cómo de peor” es nuestro peor caso.

8. CONCLUSIONES, VALORACIONES PERSONALES Y PROBLEMAS ENCONTRADOS

Como conclusión de esta práctica opinamos que finalmente, a pesar de ciertas dificultades encontradas, hemos podido completarla satisfactoriamente. Sin embargo, creemos que se podría mejorar el trabajo realizado en ciertos aspectos, como el estudio del tiempo de ejecución en el caso promedio en particular, el cual no fuimos capaces de obtener de forma satisfactoria. Seguramente, si pudiésemos dedicarle más tiempo a la práctica, podríamos haber sacado mejores conclusiones sobre el tiempo promedio.

Además, creemos que el trabajo realizado ha sido de gran utilidad para poder comprender y afianzar conocimientos de los temas 1 y 2 de la asignatura AED2, pues creemos que es una práctica profunda y que requiere una buena comprensión de la teoría aplicada para poder completarla de forma satisfactoria. También nos ha ayudado a ver cómo esta técnica puede ser de gran utilidad para la resolución de algunos problemas de programación.

Durante el desarrollo de esta práctica nos han ido surgiendo problemas y dudas, sobre todo relacionadas con el estudio teórico del tiempo de ejecución, pues nunca habíamos trabajado con 2 variables anteriormente, las cuales en la mayoría de los casos fueron resueltas con la ayuda del profesor Norberto Marín. No obstante, en otros casos pudimos solventar el problema por nuestra cuenta tras dedicar un tiempo a reflexionar sobre él. También encontramos ayuda en el libro de la asignatura, *Algoritmos y Estructuras de Datos Volumen 2*, sobre todo en el apartado en el que se habla del estudio experimental del tiempo de ejecución.

Para finalizar, tras sumar todo el tiempo dedicado a esta práctica y a la redacción de la presente memoria, creemos que aproximadamente dedicamos unas 35 horas a su realización.

9. REFERENCIAS BIBLIOGRÁFICAS

Giménez Cánovas, D.; Cervera López. J; García Mateos, G.; Marín Pérez, N. (2003) *Algoritmos y Estructuras de Datos – Volumen II – Algoritmos.*