Memoria de la práctica 1

Divide y vencerás
Algoritmos y Estructuras de Datos II
Universidad de Murcia

Sergio Ortuño Aguilar y Juan Francisco Carrión Molina

Abril de 2019

Subgrupo 1.1
Curso 2018 – 2019

sergio.ortunoa@um.es
juanf.carrion@um.es

Índice

1. Introducción **2**

2. Diseño del algoritmo **2**

2.1. Variables globales y tipos 2

2.2. Método SolucionDirecta 2

2.3. Método Pequeno 3

2.4. Método Combinar 3

2.5. Método DivideVenceras 4

2.6. Programa principal 4

3. Estudio teórico del tiempo de ejecución **4**

3.1. Tiempo de ejecución para SolucionDirecta 5

3.2. Tiempo de ejecución para Pequeno 5

3.3. Tiempo de ejecución para Combinar 6

3.4. Tiempo de ejecución para DivideVenceras y general 6

4. Implementación **6**

5. Validación del diseño **8**

6. Estudio experimental del tiempo de ejecución **9**

6.1. Pruebas de caso mejor 9

6.2. Pruebas de caso peor 10

6.3. Pruebas de caso promedio 10

7. Contraste teórico-experimental **10**

7.1. Contraste teórico-experimental de caso mejor 11

7.2. Contraste teórico-experimental de caso peor 11

7.3. Contraste teórico-experimental de caso promedio 11

7.4. Contraste teórico-experimental general 12

8. Conclusiones **12**

1. Introducción

La utilización de la técnica divide y vencerás en el diseño de algoritmos se basa en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de tipo igual o similar hasta que estos llegan a ser lo suficientemente sencillos para resolverse directamente. Finalmente, las soluciones de los subproblemas se combinan para dar una solución al problema original.

La aplicación de esta técnica en concreto se ha entendido como un proceso metódico que se explica detalladamente a continuación. El ejercicio asignado es el siguiente:

5) Dados una cadena `A` *de longitud* `n` *y un carácter* `C`*, se trata de encontrar la subcadena de* `m` *caracteres consecutivos con más apariciones del carácter* `C` *y dar el número de veces que aparece el carácter en la subcadena. Cuando haya más de una solución, será válido devolver cualquiera de ellas.*

2. Diseño del algoritmo

Resumen

(1) Pseudocódigo y explicación del algoritmo, justificando las decisiones de diseño, las estructuras de datos y las funciones básicas del esquema algorítmico.

Se ha diseñado una solución al problema planteado mediante la técnica divide y vencerás para diseño de algoritmos. Cada problema se divide en dos subproblemas de forma recursiva.

2.1. Variables globales y tipos

En la solución diseñada no se utilizan variables globales. Los datos se pasan a los subproblemas a través de los parámetros de las funciones. Esto permite organizar mejor el código y separar los casos.

Se define un tipo abstracto de datos (TAD) `Solucion` que representa la solución a un problema o subproblema, conteniendo los atributos `posicion` y `apariciones`, tal y como nos especifica el enunciado del problema. Este TAD facilita el paso de soluciones entre funciones.

```
1 tipo Solucion {
2     entero posicion;
3     entero apariciones;
4 }
```

2.2. Método SolucionDirecta

A continuación, definimos el método `SolucionDirecta`, que equivale a la función genérica de mismo nombre en el método general de divide y vencerás y que nos permite resolver directamente problemas de tamaño pequeño.

Esta función recibe los parámetros `m` y `C` del problema original y el parámetro `A` que corresponderá a cada subproblema. Además, el parámetro `posicionRelativa` nos permite devolver la posición de la cadena solución del subproblema con respecto a la cadena del problema original.

En resumen, el comportamiento de la función `SolucionDirecta` es el siguiente. Tras inicializar las variables, el primer `para` recorre la cadena carácter a carácter hasta la posición `longitud(A) - m`. Para cada pasada, si el carácter en el que estamos coincide con `C` o si estamos en la última pasada, contamos las apariciones de dicho carácter en una subcadena de longitud `m` (mediante el segundo `para`). Por último comprobamos, mediante las variables `maximoPosicion` y `maximoApariciones` si las apariciones contadas en la pasada actual superan a las anteriores, y devolvemos una solución correspondiente a la subcadena con más apariciones, identificada por su posición y el número de apariciones.

```
funcion SolucionDirecta (cadena A, entero m, caracter C, entero posicionRelativa):
    Solucion {
    entero maximoPosicion := -1;
    entero maximoApariciones := -1;

    para entero i := 0 hasta (longitud(A) - m) hacer:
        entero apariciones := 0;
        si (A[i] = C) o (i = longitud(A) - m) entonces:
            para entero j := 0 hasta (m - 1) hacer:
                si A[i + j] = C entonces:
                    apariciones := apariciones + 1;
                finsi;
            finpara;
            si apariciones > maximoApariciones entonces:
                maximoPosicion := i;
                maximoApariciones := apariciones;
            finsi;
        finsi;
    finpara;

    Solucion s;
    s.posicion := maximoPosicion + 1 + posicionRelativa;
    s.apariciones := maximoApariciones;

    devolver s;
}
```

2.3. Método Pequeno

La función `Pequeno`, que equivale a la función genérica de mismo nombre en el método general de divide y vencerás, nos permite comprobar si un problema es suficientemente pequeño como para ser resuelto directamente.

En nuestro caso, un problema es suficientemente pequeño si la longitud de la cadena dada es inferior a dos veces la longitud de la cadena que buscamos.

```
funcion Pequeno (cadena A, entero m): booleano {
    si longitud(A) >= 2 * m entonces:
        devolver falso;
    si_no:
        devolver verdadero;
    finsi;
}
```

2.4. Método Combinar

A continuación, definimos la función `Combinar`, que equivale a la función genérica del mismo nombre en el método general de divide y vencerás y que nos permite obtener la solución al problema original a partir de las soluciones de los subproblemas.

Esta función recibe como parámetros, por un lado, dos tipos `Solucion` y, por otro lado, la cadena `A`, el entero `m`, el carácter `C` y el entero `posicionRelativa`. Estos últimos cuatro parámetros nos permiten, además de combinar las soluciones de los subproblemas, buscar una solución en la frontera entre ambos.

Definimos frontera como la cadena formada por los últimos `m - 1` caracteres del primer subproblema y los primeros `m - 1` caracteres del segundo subproblema. Esta búsqueda de solución al problema se realiza aquí ya que en el método `SolucionDirecta` no es conveniente. Tras obtener la solución de la frontera, compara las tres soluciones y devuelve la de más apariciones.

```
funcion Combinar (Solucion s1, Solucion s2, cadena A, entero m, caracter C, entero
    posicionRelativa): Solucion {
    posicionRelativa := posicionRelativa + redondear(longitud(A) / 2) - m + 1;
    cadena frontera := subcadena(A, redondear(longitud(A) / 2) - m + 1, (m - 1) * 2);
    Solucion s3 := SolucionDirecta(frontera, m, C, posicionRelativa);

    si s1.apariciones > s2.apariciones entonces:
        s2 = s1;
    finsi;
    si s2.apariciones > s3 apariciones entonces:
        devolver s2;
    si_no:
        devolver s3;
    finsi;
}
```

2.5. Método DivideVenceras

Finalmente, la función `DivideVenceras` es el método principal para aplicar la resolución de un problema mediante la técnica de diseño aplicada. Esta función recibe los parámetros `A`, `,` y `C` del problema inicial, así como un parámetro `posicionRelativa` que en caso del problema inicial será `0`.

El método comprueba si el problema es lo suficientemente pequeño y, en este caso, lo resuelve mediante la función `SolucionDirecta`. En caso contrario, realiza el trabajo correspondiente a la función genérica "Dividir"del método general de divide y vencerás, que en nuestro caso se realiza directamente generando dos subcadenas de longitud igual a la mitad de la original. Para terminar, devuelve la combinación entre las soluciones obtenidas de los dos subproblemas.

```
funcion DivideVenceras (cadena A, entero m, caracter C, entero posicionRelativa):
    Solucion {
    si noDefinido(posicionRelativa) entonces:
        posicionRelativa := 0;
    fin_si;

    si Pequeno(A, m) entonces:
        devolver SolucionDirecta(A, m, C, posicionRelativa);
    si_no:
        entero mitad := redondear(longitud(A) / 2);
        cadena A1 := subcadena(A, 0, mitad);
        cadena A2 := subcadena(A, mitad, mitad);

        devolver Combinar(DivideVenceras(A1, m, C, posicionRelativa),
                          DivideVenceras(A2, m, C, posicionRelativa + mitad),
                          A, m, C, posicionRelativa);
    finsi;
}
```

2.6. Programa principal

En el programa principal, adaptado para resolver los casos de prueba, se definen los parámetros y se llama a este último método.

3. Estudio teórico del tiempo de ejecución

Resumen

(2) Estudio teórico del tiempo de ejecución del algoritmo (t_m, t_M y t_p) y obtención de conclusiones acerca de los órdenes.

El tiempo de ejecución del algoritmo se ha estudiado a partir del planteamiento en pseudocódigo. Para simplificarlo, hemos dividido el análisis en varias partes correspondientes a los distintos métodos.

En el algoritmo diseñado, los tiempos de ejecución pueden depender de `n` = *longitud*(A), de `m` y del contenido de `A`, es decir, la entrada de datos. En los casos en los que el tiempo de ejecución dependa de la entrada de datos, el tiempo promedio se calcula a través de la probabilidad de los casos mejor y peor.

En el conteo de instrucciones, establecemos las siguientes consideraciones:

- Asignaciones constantes, *+1*.
- Comprobación del `si` (`if`), incluido el `si_no` (`else`), *+1*.
- Comprobación del `para` (`for`) (dentro del bucle), *+1*.
- Inicialización e incremento del `para` (`for`), *+0*.
- Reserva del tipo abstracto de datos `Solucion`, *+2*.

3.1. Tiempo de ejecución para SolucionDirecta

En el caso de la función `SolucionDirecta`, el tiempo de ejecución, t_{SD} depende de la entrada de datos, por lo que estudiamos los casos mejor, t_{SD_m}; peor, t_{SD_M}; y promedio, t_{SD_p}.

$$t_{SD_m}(n,m) = 2 + \sum_{i=0}^{n-m}(3) + 4 = 3n - 3m + 8$$

$$t_{SD_M}(n,m) = 2 + \sum_{i=0}^{n-m}(3 + \sum_{j=0}^{m}(2) + 1) + 2 + 4 = 2mn - 2m^2 - 4m + 6n + 14$$

$$t_{SD_p}(n,m) = 2 + \sum_{i=0}^{n-m}(3 + (1 - \frac{n-m}{n-m+1}\frac{25}{26})(\sum_{j=0}^{m}(2 + \frac{1}{26}1) + 1 + (\frac{1}{26})^2 2)) + 2 + 4$$

Aunque se ha hecho en papel, no se da una simplificación de $t_{SD_p}(n,m)$ por aproximarse a t_{SD_m}, lo cual se expresará en la conclusión final.

Nótese que el término $(1 - \frac{n-m}{n-m+1}\frac{25}{26})$ corresponde a la probabilidad de que se cumpla la condición de la instrucción `si (A[i] = C) o (i = longitud(A) - m)` calculada a través de las *leyes de De Morgan*.

El término $\frac{1}{26}$ corresponde a la probabilidad de que se cumpla la condición de la instrucción `si A[i + j] = C`. Por último, el término $(\frac{1}{26})^2$ corresponde a la probabilidad de que se cumpla la condición de la instrucción `si apariciones > maximoApariciones`.

La conclusión obtenida es la siguiente:

$$t_{SD} \in O(mn - m^2)$$

3.2. Tiempo de ejecución para Pequeno

En el caso de la función `Pequeno`, observamos que el tiempo de ejecución, t_Q, es siempre el mismo, independientemente de la entrada de datos. La ejecución de este método se realiza en orden constante:

$$t_Q(n,m) \in O(1)$$

3.3. Tiempo de ejecución para Combinar

En el caso de la función `Combinar`, el tiempo de ejecución, t_C depende de la entrada de datos en una sola asignación constante, por lo que estudiamos solamente el caso general. La conclusión obtenida es la siguiente:

$$t_C(n,m) \in O(t_{SD}(n,m))$$

3.4. Tiempo de ejecución para DivideVenceras y general

En el caso de la función `DivideVenceras`, el tiempo de ejecución, t_{DV} no depende de la entrada de datos, por lo que estudiamos solamente el caso general. Este tiempo de ejecución combina los tiempos de ejecución de todos los métodos, siendo así el general para el algoritmo, t. Tratándose de un sistema con ecuación de recurrencia, separamos varios casos en la definición del tiempo de ejecución.

$$t(n,m) = t_{DV}(n,m) = \begin{cases} 2 + t_Q(n,m) + t_{SD}(n,m), & \text{si } \frac{n}{m} < 2 \\ 2 + t_Q(n,m) + 3 + 2t_{DV}(\frac{n}{2},m) + t_C(n,m), & \text{si } \frac{n}{m} \geq 2 \end{cases}$$

$$t(n,m) = \begin{cases} 3 + mn - m^2, & \text{si } \frac{n}{m} < 2 \\ 6 + 2t(\frac{n}{2},m) + mn - m^2, & \text{si } \frac{n}{m} \geq 2 \end{cases}$$

Para resolver la ecuación de recurrencia con dos parámetros, extraemos la `m` como constante y la fijamos a dos valores, 100 y 1000, los mismos que utilizaremos para la experimentación. Obtenemos estos resultados, que nos llevan a las conclusiones finales.

$$t(n,100) \in O(100n - 10000)$$
$$t(n,1000) \in O(1000n - 1000000)$$

$$\textit{Entonces, } t(n,m) \in O(mn - m^2)$$

Como conclusión, observamos que el algoritmo de divide y vencerás consigue resolver el problema en un tiempo de orden igual al algoritmo de resolución directa.

4. Implementación

Resumen

(3) Programación del algoritmo (lo normal es hacer el programa tras haber diseñado y estudiado teóricamente el algoritmo). El programa debe ir documentado, con explicación de qué es cada variable, qué realiza cada función y su correspondencia con las funciones básicas del esquema algorítmico correspondiente.

La implementación del algoritmo se ha realizado en lenguaje C++ en un único archivo `main.cpp` con todas las funciones.

Se utiliza la librería estándar (STD) de C++ en el programa. Concretamente, los componentes `iostream` para operaciones de entrada y salida, `string` para operaciones con cadenas de texto y `math.h` para operaciones matemáticas básicas.

El tipo `Solucion` se implementa como una clase de C++ con dos atributos correspondientes a los establecidos en el pseudocódigo. El resto de métodos se implementan, también, de forma similar al pseudocódigo, con algunas pequeñas excepciones relativas a particularidades del lenguaje. La explicación de qué es cada variable, qué realiza cada función y su correspondencia con las funciones básicas del esquema de divide y vencerás son equivalentes a las del apartado Sección 2 Diseño del algoritmo.

Destacamos la inicialización del parámetro `posicionRelativa` del método `DivideVenceras`, que en la implementación se denomina `ini` y que se sobrecarga en la cabecera de dicho método. En las operaciones matemáticas de división, especialmente en la creación de los subproblemas, se utiliza una aproximación en los casos en que la `n` sea impar.

```
#include <iostream>
#include <string>
#include <math.h>

using namespace std;

class Solucion {
    public:
        int posicion;
        int apariciones;
};

Solucion SolucionDirecta(string A, int m, char C, int ini) {
    int maxposicion;
    int maxapariciones = -1;

    for (int i = 0; i <= (A.length() - m); i++) {
        int apariciones = 0;
        if ((A[i] == C || i == A.length() - m)) {
            for (int j = 0; j < m; j++) {
                if (A[i + j] == C) {
                    apariciones++;
                }
            }

            if (apariciones > maxapariciones) {
                maxposicion = i;
                maxapariciones = apariciones;
            }
        }
    }

    Solucion s;
    s.posicion = maxposicion + 1 + ini;
    s.apariciones = maxapariciones;

    return s;
}

bool Pequeno(string A, int m) {
    int n = A.length();
    if (n >= 2 * m) {
        return false;
    } else {
        return true;
    }
}

Solucion Combinar(Solucion s1, Solucion s2, string A, int m, char C, int
ini) {
```

```
51      ini = ini + ceil(float(A.length()) / 2) - m + 1;
52      string frontera = A.substr(ceil(float(A.length()) / 2) - m + 1, (m - 1) * 2);
53      Solucion s3 = SolucionDirecta(frontera, m, C, ini);
54
55      if (s1.apariciones > s2.apariciones) {
56          s2 = s1;
57      }
58
59      if (s2.apariciones > s3.apariciones) {
60          return s2;
61      } else {
62          return s3;
63      }
64  }
65
66  Solucion DivideVenceras(string A, int m, char C, int ini = 0) {
67      if (Pequeno(A, m)) {
68          return SolucionDirecta(A, m, C, ini);
69      } else {
70          int lon = ceil(float(A.length()) / 2);
71          string A1 = A.substr(0, lon);
72          string A2 = A.substr(lon, A.length() / 2);
73
74          return Combinar(DivideVenceras(A1, m, C, ini),
75                          DivideVenceras(A2, m, C, ini + lon),
76                          A, m, C, ini);
77      }
78  }
79
80  int main() {
81      string A;
82      int m, ncasos;
83      char C;
84
85      cin >> ncasos;
86
87      for (int i = 0 ; i < ncasos; i++){
88          cin >> C;
89          cin >> m;
90          cin >> A;
91
92          Solucion s = DivideVenceras(A, m, C);
93
94          cout << "Caso: " << i + 1 << endl;
95          cout << "Posicion: " << s.posicion << endl;
96          cout << "Apariciones: " << s.apariciones << endl;
97      }
98
99      return 0;
100 }
```

5. Validación del diseño

Resumen

(4) Validación del algoritmo, justificando los experimentos realizados para asegurar que el algoritmo funciona correctamente y, en su caso, programas utilizados para la validación.

Para validar el algoritmo construido hemos estudiado detenidamente los tipos de casos que podrían darse a partir de los datos de entrada. Hemos validado la salida del método de solución directa para varios casos haciendo uso de la especificación del enunciado.

Una vez hecho esto, hemos comprobado que las soluciones resultantes con nuestro algoritmo de divide y vencerás son equivalentes a las que aparecerían si resolviésemos los problemas a través del método de solución directa. De esta manera, se comprueba que las soluciones de los subproblemas se obtienen y combinan correctamente. Lo mismo pasa para las soluciones en las fronteras de los subproblemas.

Se han realizado pruebas con cadenas de distinto contenido y longitud, así como casos extremos para validar definitivamente el algoritmo. Los experimentos concretos se han generalizado en los tipos de cadenas que producen los generadores del apartado Sección 6 Estudio experimental del tiempo de ejecución.

6. Estudio experimental del tiempo de ejecución

Resumen

(5) Estudio experimental del tiempo de ejecución para distintos tamaños de problema. Habrá que experimentar con tamaños suficientemente grandes para obtener resultados significativos. En el caso de resolver la opción a) habrá que estudiar cómo influye en el tiempo de ejecución hacer 2 o 3 subdivisiones.

Para realizar el estudio experimental del tiempo de ejecución construimos distintos generadores de casos de prueba. Teniendo en cuenta que el tiempo de ejecución de nuestro algoritmo variará según la entrada de datos, construimos generadores para los casos mejor, peor y promedio.

Utilizamos las siguientes funciones comunes para los generadores de casos de prueba.

- `funcion generarCaracter(): caracter` devuelve un carácter aleatorio del alfabeto `(a - z)`.
- `funcion generarEntero(entero longitudMaxima): entero` devuelve un número entero entre 1 y la longitud máxima de la cadena.
- `funcion generarCadena(...): cadena` devuelve una cadena según las necesidades y parámetros de cada grupo de casos.

Los generadores se pondrán en acción en el apartado Sección 7 Contraste teórico-experimental, con los resultados en representación gráfica, para simplificar el análisis. Lo que se presenta a continuación son las bases de los generadores, asumiendo que en la implementación final de cada uno se incluye la lectura del número de casos y de la longitud máxima, el bucle de generación de elementos y la impresión de los casos de prueba.

6.1. Pruebas de caso mejor

En cada caso mejor, generaremos una `m` y una cadena `A` de `n` = $longitud(A)$ y contenido de caracteres en los que nunca aparezca el carácter `C`. Este carácter se genera previamente en cada caso y se pasa como parámetro `excepcion` a `generarCadena`. Nuestro generador de cadenas de prueba es el siguiente:

```
1 funcion generarCadena(entero longitudMaxima, caracter excepcion): cadena {
2     cadena A;
3     entero longitud := generarEntero(longitudMaxima);
4 
5     para entero z := 1 hasta longitud hacer:
6         caracter generado := generarCaracter();
7 
8         mientras generado = excepcion hacer:
```

```
            generado := generarCaracter();
        finmientras;
        A := A + generado;
    finpara;
}
```

6.2. Pruebas de caso peor

En cada caso peor, generaremos una `m` y una cadena `A` de `n` = $longitud(A)$ y contenido de caracteres en los que únicamente aparezca el carácter `C`. Este carácter se genera previamente en cada caso y se pasa como parámetro `repetido` a `generarCadena`. Nuestro generador de cadenas de prueba es el siguiente:

```
funcion generarCadena(entero longitudMaxima, caracter repetido): cadena {
    cadena A;
    entero longitud := generarEntero(longitudMaxima);

    para entero z := 1 hasta longitud hacer:
        A := A + repetido;
    finpara;
}
```

6.3. Pruebas de caso promedio

En cada caso peor, generaremos una `m` y una cadena `A` de `n` = $longitud(A)$ y contenido de caracteres aleatorios. Nuestro generador es el siguiente:

```
funcion generarCadena(entero longitudMaxima): cadena {
    cadena A;
    entero longitud := generarEntero(longitudMaxima);

    para entero z := 1 hasta longitud hacer:
        caracter generado := generarCaracter();
        A := A + generado;
    finpara;
}
```

7. Contraste teórico-experimental

Resumen

(6) Contraste del estudio teórico y el experimental, buscando justificación a las discrepancias entre los dos estudios.

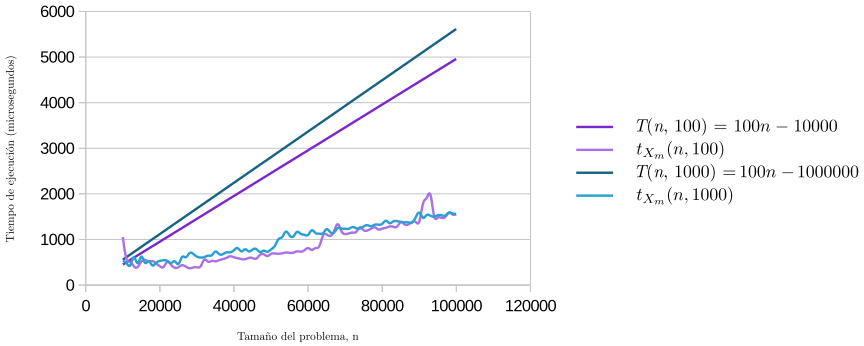
Previamente a poner en común los resultados de los estudios teórico y experimental, ejecutamos los generaciones de casos de prueba. Fijamos la `n` en un rango de 10000 a 100000 y la `m` en dos valores, 100 y 1000, para poder realizar la representación gráfica en dos dimensiones.

Recogemos los datos de tiempo experimentales ($t_{X_m}(n, 100)$, $t_{X_m}(n, 1000)$, $t_{X_M}(n, 100)$, $t_{X_M}(n, 1000)$, $t_{X_p}(n, 100)$ y $t_{X_p}(n, 1000)$), en varias gráficas y los superponemos con el orden teórico obtenido para $t(n, m) \in O(mn - m^2)$. Este orden lo representamos con la función `T` que corresponde al mismo con una constante multiplicativa para ajustar la altura de la función al tiempo experimental.

Destacar que, en cada una de las siguientes gráficas, la escala de tiempo de ejecución es diferente, pero la función `T` que da el orden es la misma en todos los casos.

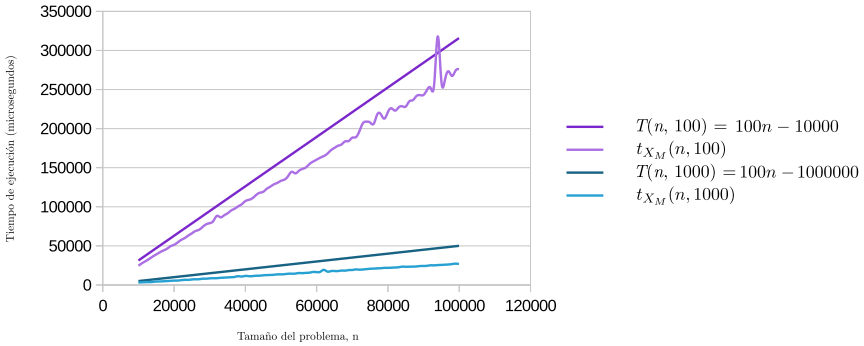
7.1. Contraste teórico-experimental de caso mejor

A continuación presentamos una gráfica con la representación de los tiempos de ejecución experimentales $t_{X_m}(n, 100)$, $t_{X_m}(n, 1000)$ y teóricos $t(n, 100) \in O(100n - 10000)$, $t(n, 1000) \in O(1000n - 1000000)$.



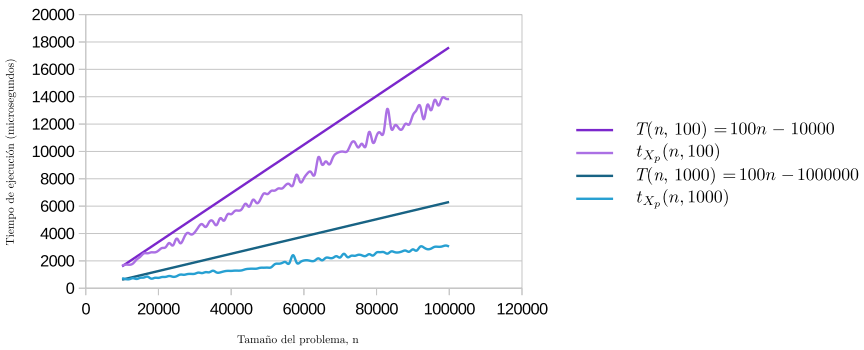
7.2. Contraste teórico-experimental de caso peor

A continuación presentamos una gráfica con la representación de los tiempos de ejecucióne experimentales $t_{X_M}(n, 100)$, $t_{X_M}(n, 1000)$ y teóricos $t(n, 100) \in O(100n - 10000)$, $t(n, 1000) \in O(1000n - 1000000)$.



7.3. Contraste teórico-experimental de caso promedio

A continuación presentamos una gráfica con la representación de los tiempos de ejecución experimentales $t_{X_p}(n, 100)$, $t_{X_p}(n, 1000)$ y teóricos $t(n, 100) \in O(100n - 10000)$, $t(n, 1000) \in O(1000n - 1000000)$.



7.4. Contraste teórico-experimental general

La conclusión sobre el contraste de los estudios teórico y experimental del tiempo de ejecución es que el orden obtenido, $O(nm - m^2)$, representa, como esperábamos, una cota superior del tiempo de ejecución real del algoritmo. Esto nos permite saber qué recursos máximos va a necesitar el algoritmo para ejecutarse.

8. Conclusiones

Resumen

Conclusiones y valoraciones personales de la actividad y estimación del tiempo total que se ha tardado en completarla, distinguiendo entre tiempo dedicado a la versión básica y a los apartados opcionales.

La realización de esta práctica ha supuesto todo un reto para los dos componentes del equipo que la ha realizado.

Comprender el problema enunciado y resolverlo aplicando los distintos pasos de diseño del algoritmo ha llevado al planteamiento de muchas situaciones difíciles, en las que se ha recurrido al trabajo intensivo en equipo y a la búsqueda e investigación de soluciones en distintas fuentes, como el libro *Algoritmos y Estructuras de Datos*[1].

La separación del proyecto en apartados ha facilitado la resolución del ejercicio, que igualmente ha requerido para su terminación todas las sesiones de laboratorio correspondientes y bastantes más horas de trabajo en casa. En total, calculamos haber dedicado aproximadamente 5 horas presenciales y 20 de trabajo autónomo para la realización de la práctica en grupo.

Obtener un primer algoritmo de solución directa fue relativamente fácil. Construir el algoritmo con la técnica de divide y vencerás ha supuesto un esfuerzo considerable, que se ha completado con las distintas mejoras propias, como la reducción de pasadas de los bucles.

La complicación real ha venido a la hora de analizar el algoritmo construido y obtener una cota sobre el tiempo de ejecución. Finalmente, conseguimos completar toda la práctica satisfactoriamente y con los resultados esperados.

"El motor analítico no ocupa un terreno común con las máquinas calculadoras. Mantiene una posición totalmente propia y las consideraciones que sugiere son más interesantes en su naturaleza."

Ada Lovelace

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