Universidad de Murcia
Murcia, Abril 2019

Algoritmos y Estructuras de Datos II

Práctica 1: Aplicación del esquema algorítimico de Divide y Vencerás

Alumno: Mourad Abbou Azaz

Email: mourad.abbou@um.es

Grupo: 3.1

Tutor: Domingo Giménez Cánovas

Departamento de Informática y Sistemas

Ingeniería Informática

Facultad de Informática

Contents

1 Diseño y explicación del algoritmo

El ejercicio asignado nos pide que diseñemos un programa tal que, dadas dos cadenas, encontrar la secuencia de vocales más larga que tienen en común, y devolver su tamaño y la posición en la que se encuentra dicha secuencia. Ambas cadenas tienen que tener el mismo tamaño.

1.1 Decisiones de diseño

El planteamiento que se ha tomado para tratar el problema ha sido dividiendo, de manera recursiva, la cadena en dos hasta que su longitud se pueda tratar con el caso base. Para cada parte calculamos la secuencia de vocales más larga y luego hacemos una combinación.

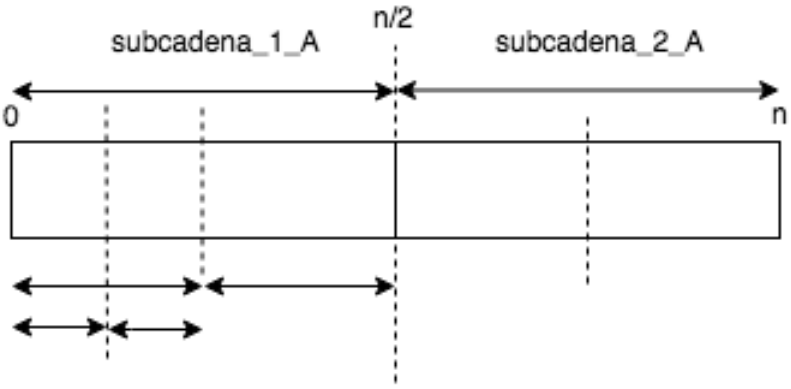


Figure 1.1: Esquema de partición de las cadenas

Como caso base, se ha optado por tratar el problema de manera directa cuando la longitud de una cadena es igual a 1.

Como estructura de datos se ha decidido utilizar un array con dos posiciones:

- La primera posición contiene el tamaño de la secuencia más larga de vocales
- Las segunda posición contiene la posición de la secuencia más larga de vocales en la cadena principal

Como solución directa al caso base (n = 1), se ha propuesto esta función:

```
int[] solucion_directa(string cadena1, string cadena2, int posicion) {
    int resultado[] = {0, posicion}
    if (es_vocal(cadena1[0]) && es_vocal(cadena2[0]))
        resultado[0] = 1  ;
```

```
        return resultado;
    }
```

En dicha función, al tener tamaño 1, se comprueba si solo el carácter de cada cadena es vocal. Si lo son, se devuelve el resultado con la posición en la que se encuentran.

1.2 Pseudocódigo y explicación de las funciones

```
int algoritmo(cadena1: String, cadena2: String, pos_inicial: integer, n: integer);
```

Esta es la función que ejecuta el algoritmo basado en el esquema de *Divide y Vencerás*.
Los argumentos que se toman son:

- **cadena1** : Primera cadena con la que se va a operar.
- **cadena2** : Segunda cadena con la que se va a operar.
- **pos_inicial**: Esta es la posición en la que comienza la cadena con respecto a la principal. El motivo de este argumento es el control del comienzo de la subcadena que tenga más vocales. Por ejemplo:
  Dada esta cadena principal:

  a b d d o i h g y i u j k z o i u d

  Si cogemos la subcadena:

  d o i h y i

  Su posición inicial es 3, ya que comienza en la posición 3 de la cadena principal.
- **n** : Es la longitud de las cadenas. Ambas cadenas deben tener la misma longitud

1.2.1 Algoritmo

```
1 int[] algoritmo(cadena1: String, cadena2: String, pos_inicial: integer, caso_base: integer, n: integer):
2 begin
3     int resultado[] := {0, 0}
4     if (n <= caso_base)
5     then
6         return solucion_directa(cadena1, cadena2, pos_inicia, n);
7     end_if
8
```

```
9     subcadena_derecha_1 := sub_string(cadena1, 0, n / 2);
10    subcadena_izquierda_1 := sub_string(cadena1, n/2, n);
11    subcadena_derecha_2 := sub_string(cadena2, 0 n/2);
12    subcadena_izquierda_2 := sub_string(cadena2, 0, n/2);
13
14    int parte_derecha[] := algoritmo(subcadena_derecha_1,
15                                     subcadena_derecha_3,pos_inicial, n/2);
16    int parte_izquierda[] := algoritmo(subcadena_derecha_2,
17                                     subcadena_izquierda_2, n/2, n/2);
18
19    # A partir de aqui, comienza la funcin de combinacin
20    if(hay_vocales_cosecutivas(subcadena_derecha_1, subcadena_izquierda_1))
21    then
22        int parte_ms_derecha[] := get_parte_ms_derecha(subcadena_derecha_1,
23                                     subcadena_izquierda_1, pos_inicial, n/2);
24        int parte_ms_izquierda[] := get_parte_ms_izquierda(subcadena_izquierda_1,
25                                     subcadena, subcadena_izquierda_2, n/2, n/2);
26
27        if (parte_ms_derecha[0] + parte_ms_izquierda[0] > max(parte_derecha[0], parte_izquierda[0])
28        then
29            resultado[0] := parte_ms_derecho[0] + parte_ms_izquierdo[0];
30            resultado[1] := parte_ms_derecho[1];
31        else
32            if (parte_derecha[0] >= parte_izquierda[0])
33            then
34                resultado[0] := parte_derecha[0];
35                resultado[1] := parte_derecha[1];
36            else
37                resultado[0] := parte_izquierda[0];
38                resultado[1] := parte_izquierda[1];
39            end_if
40        end_if
41    else
42        if (parte_derecha[0] >= parte_izquierda[0])
43        then
44            resultado[0] := parte_derecha[0];
45            resultado[1] := parte_derecha[1];
46        else
47            resultado[0] := parte_izquierda[0];
48            resultado[1] := parte_izquierda[1];
49        end_if
50    end_if
```

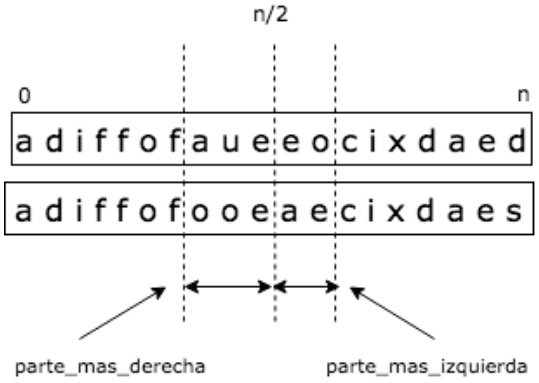
```
51      return resultado;
52  end
```

Comenzamos inicializando nuestro array `resultado`. Como caso base, si la longitud de ambas cadenas es 1, se invoca a la función `solucion_directa()`. En caso contrario, se divide cada cadena en 2 partes, y a continuación:

1. Se calcula cada parte (`parte_derecha[]` y `parte_izquierda[]`) de manera recursiva llamando a la misma función en la línea 14 y 16.

2. Para el tratamiento de la frontera entre cadenas, se comprueba si hay vocales consecutivas entre la última letra de la primera parte y la primera de la segunda. En ese caso:

   (a) Se calcula la parte más derecha y más izquierda donde hay vocales. En este caso se realiza desde n/2 hacia la izquierda o la derecha, hasta que termine la secuencia de vocales llamando a las funciones `get_parte_más_derecha()` y `get_parte_más_izquierda()` respectivamente. Se estima que ambas tienen un tiempo de ejecución de $\frac{n}{2}$ cada una.



   (b) Ambas partes se unen y se compara con el resto de resultados de las subcadenas. En caso de que la unión sea mayor que dichas subcadenas, se inserta su resultado y su posición en la cadena principal (línea 29 y 30).

3. En caso de que el resultado de la unión de secuencias sea menor los resultados de los algoritmos de las subcadenas calculado previamente, se comparan dichos resultados entre sí y se devuelve el de mayor valor.

4. Se devuelve el array de resultados.

Las funciones que aparecen en el algoritmo son:

- `solucion_directa()`: Es la función que calcula la solución directa cuando se da el caso base.

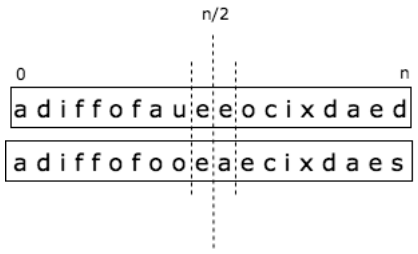
```
1  int[] solucion_directa(cadena1: String, cadena2: String, pos_inicial: integer, tam_caso_base: integer)
2  begin
```

```
3     int[] resultado = {0, 0}
4     int contador_vocales := 0;
5     for i = 0 to tam_caso_base
6     do
7         if (es_vocal(cadena1[i]) AND es_vocal(cadena2[i]))
8         then
9             contador_vocales := contador_vocales + 1;
10        end_if
11    end_for
12    resultado[0] := contador_vocales;
13    resultado[1] := pos_inicial;
14    return resultado;
15 end
```

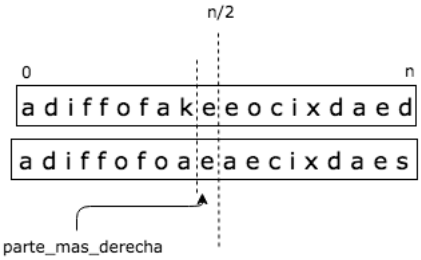
En nuestro caso, se ha elegido como caso base una cadena de tamaño 1 para el estudio teórico.

- `sub_str(String cadena, pos_inicial, longitud)`: Dada una cadena, esta función devuelve una subcadena tomando una posición inicial y la longitud que va a tener. En C++, esta función consume un tiempo de orden $O(n)$, siendo n la longitud de la cadena.

- `hay_vocales_consecutivas(String, String)`: Es una función que trata la frontera entre dos cadenas consecutivas que comprueba que la última letra del primer argumento y la primera letra del segundo argumento sean vocales.



- `calcular_parte_más_derecha()`: Dada una subcadena, intenta calcular su parte más derecha y cercana al final de la cadena.



- `calcular_parte_más_izquierda()`: Dada una subcadena, intenta calcular su parte más

izquierda y cercana al final de la cadena. Tanto esta función como la anterior son funciones auxiliares que se utilizar dentro del marco de combinación de soluciones.

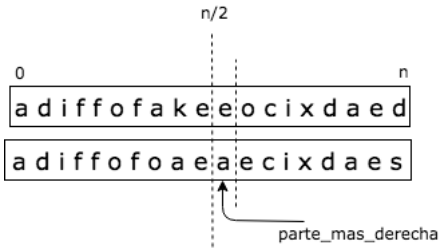


Figure 1.2: Caption

2 Estudio teórico del algoritmo diseñado

Tras diseñar el algoritmo, se ha realizado el estudio teórico de este, mediante el conteo de instrucciones, para calcular su tiempo más favorable, desfavorable y promedio dado un número n suficientemente grande.

2.1 Estudio teórico del caso base

Antes de realizar el estudio general, se realiza un estudio teórico del caso base para conocer su conocimiento en caso de elección de un determinado tamaño para la solución directa.

2.1.1 Caso más favorable

El tiempo en el mejor caso se da cuando las cadenas del caso base están formadas solo por consonantes.

$$t_m(n)_{casobase} = 1 + 1 + \sum_{i=1}^{n} 1 + 3$$
$$t_m(n)_{casobase} = 5 + n \in O(n)$$

En cuanto al conteo de instrucciones, tomando como referencia el pseudocódigo de la solución directa:

- Las dos primeras instrucciones corresponden a las líneas 3 y 4.
- En las línea 5 hay un bucle *for*, lo que corresponde con un sumatorio.
- Al ser todo consonantes solo se cuenta la condición en la línea 7.
- Las líneas 13, 14 y 15 corresponden a las 3 últimas instrucciones.

2.1.2 Caso más desfavorable

El tiempo en el peor caso se da cuando las cadenas del caso base están formadas solo por vocales.

$$t_M(n)_{casobase} = 1 + 1 + \sum_{i=1}^{n} (1 + 1) + 3$$
$$t_M(n)_{casobase} = 5 + 2n \in O(n)$$

La variación es que, si las cadenas están formadas solo por vocales, la condición se cumple y se añade otra instrucción

2.1.3 Caso promedio

En el tiempo promedio, hay que tener en cuenta de que el carácter actual de la primera cadena tiene una probabilidad aproximada de $\frac{1}{5}$ de que sea vocal. Idem para el carácter actual de la segunda vocal. Por lo que la probabilidad de que ambos caracteres de cada cadena sean vocales es de $\frac{1}{5} \cdot \frac{1}{5}$

$$t_p(n)_{casobase} = 1 + 1 + \sum_{i=1}^{n}(1 + \tfrac{1}{5} \cdot \tfrac{1}{5}) + 3$$
$$t_p(n)_{casobase} = 5 + \tfrac{26}{25}n \in O(n)$$

Como se puede ver, el crecimiento en los tres casos es lineal de orden n.

Tiempo (milis.)	**O(n)**	$\Omega(n)$	$\Theta(n)$	$o(n)$
Tiempo en el mejor caso	n	n	n	$1 \cdot n$
Tiempo en el peor caso	n	n	n	$2 \cdot n$
Tiempo promedio	n	n	n	$1,04 \cdot n$

2.2 Estudio teórico general

2.2.1 Caso más favorable

$$t(n) = 1 + 1 + 4n + t(\tfrac{n}{2}) + t(\tfrac{n}{2}) + 1 + 1 + 3 + 1$$

Si volvemos a la sección del algoritmo, vemos que:

- Las dos primeras instrucciones corresponden a la linea 3 y 4.
- La función `sub_string()` tiene un tiempo de ejecución igual a n, por lo que desde la línea 9 hasta la línea 12 se ejecutan 4 instrucciones de dicha función. Dicha función existe en las librerías de C++ y tiene un orden lineal. (Más información en StackOverFlow: Substring complexity)
- $t(\frac{n}{2})$ corresponde a las llamadas del algoritmo con utilizando subcadenas de longitud $n/2$ (lineas 14 y 16).
- Tenemos la instrucción que comprueba si hay vocales consecutivas (línea 27).
- Tenemos la instrucción que comprueba si el resultado de parte derecha es mayor que de la parte izquierda (línea 42).
- Tenemos 3 instrucciones equivalentes a la asignación de resultados (líneas 34 y 42).
- Tenemos la instrucción de retorno de array (línea 51).

Para calcular la ecuación característica de $t(n)$, resolvemos la función.

$$t(n) = 1 + 1 + 4n + t(\frac{n}{2}) + t(\frac{n}{2}) + 1 + 1 + 2 + 1$$

$$t(n) = 7 + 4n + 2t(\frac{n}{2})$$

$$n = 2^k$$

$$t_k = 7 + 4 * 2^k + 2t_k$$

Ecuación caracterísitca:

$$t_k - 2t_{k-1} = 7 + 4 * 2^k$$

$$(x - 2)(x - 2)(x - 1) = 0$$

Soluciones: $x_1 = 2$, $x_2 = 2$ y $x_3 = 1$

$$t_k = C_1 * 2^k + C_2 * k * 2^k + C_3 * 1^k$$

$$k = log_2(n)$$

$$t(n) = C_1 * 2^{log_2(n)} + C_2 * log_2(n) * 2^{log_2(n)} + C_3 * 1^{log_2(n)}$$

$$t(n) = C_1 * n^{log_2(2)} + C_2 * log_2(n) * n^{log_2(2)} + C_3$$

$$t(n) = C_1 * n + C_2 * log_2(n) * n + C_3$$

Casos base:

- Para un tamaño n igual a cero, solo se ejecutarán 2 instrucciones del algoritmo general (Línea 3 y 4). Por lo que $t(0) = 2$
- Para un n igual a 1, se ejecutarán las dos primeras instrucciones mas el tiempo del caso base. El crecimiento del caso base es de orden O(n), por lo que diremos que $t(n) = n$ aproximadamente. Por lo que $t(1) = 2 + t_{caso-base}(1) = 2 + 1 = 3$

$$t(0) = 2;\ t(1) = 3;$$

Para los casos base, cuando la longitud de la cadena el 0, no se ejecuta el algoritmo. Cuando la cadena tiene tamaño 1, devuelve 6 instrucciones que consume la solución directa.

Si $n = 0$:

$$t(0) = C_1 * 0 + C_2 * log_2(0) * 0 + C_3$$

$$t(0) = C_3 = 2$$

Si $n = 1$:

$$t(1) = C_1 * 1 + C_2 * log_2(1) * 1 + C_3 = 3$$

$$t(1) = C_1 + C_3 = 3$$

$$C_1 = 1$$

Si $n = 2$

$$t(2) = 7 + 4 * (2) + 2t(\frac{2}{2})$$

$$t(2) = 7 + 8 + 2 * 3 = 21$$

$$t(2) = C_1 * 2 + C_2 * log_2(2) * 2 + C_3 = 21$$
$$t(2) = 1 * 2 + C_2 * log_2(2) * 2 + 2 = 21$$
$$t(2) = 2 + C_2 * 2 + 2 = 21$$
$$2 * C_2 = 17$$
$$C_2 = 8,5$$

Función de complejidad: $t(n) = 6 * n + 7,5 * log_2(n) * n \in \Theta(log_2 n * n)$

$$t(n) = \begin{cases} n = 0 & entonces \quad 2 \\ n = 1 & entonces \quad 3 \\ n > 1 & entonces \quad n + 8,5 * log_2(n) * n + 2 \end{cases}$$

2.2.2 Caso más desfavorable

$$t(n) = 1 + 1 + 4n + t(\tfrac{n}{2}) + t(\tfrac{n}{2}) + 1 + \tfrac{n}{2} + \tfrac{n}{2} + 1 + 3 + 1$$
$$t(n) = 8 + 5n + 2 * t(\tfrac{n}{2})$$

Las variaciones del conteo en el caso más desfavorable que se presentan con respecto al más favorable son, en la linea 22 y 24, cuando existen vocales consecutivas entre dos subcadenas, se busca la secuencia de vocales más a la izquierda de la primera subcadena con respecto a la posición n/2 y la secuencia más a la derecha de la segunda subcadena. Estas dos funciones tienen un tiempo de ejecución de $\frac{n}{2}$ cada una, por lo que las dos tienen un coste total de n.
Ecuación característica:

$$t(n) = 8 + 5n + 2t(\tfrac{n}{2})$$
$$n = 4^k$$
$$t_k = 8 + 4 * 2^k + t_{k-1}$$
$$t_k - 2t_{k-1} = 8 + 5 * 2^k$$
$$t(k) = C_1 * 2^k + C_2 * k * 2^k + C_3 * 1^k$$
$$k = log_2\ n$$
$$t(n) = C_1 * 2^{log_2\ n} + C_2 * log_2(n) * 2^{log_2\ n} + C_3 * 1^{log_2\ n}$$
$$t(n) = C_1 * n^{log_2\ 2} + C_2 * log_2(n) * n^{log_2\ 2} + C_3 * n^{log_2\ 1}$$
$$t(n) = C_1 * n + C_2 * log_2 n * n + C_3$$

Función de tiempo:

$$t(0) = 2$$
$$t(0) = C_3 = 2$$

$$t(1) = 3$$

$$t(1) = C_1 * 1 + C_2 * 0 * 1 + C_3 = 3$$

$$C_1 = 1$$

$$t(2) = 8 + 5 * 2 + 2 * t(1)$$

$$t(2) = 8 + 10 + 6 = 24$$

$$t(2) = 2 + C_2 * 2 + 2 = 24$$

$$t(2) = 4 + 2 * C_2 = 24$$

$$2 * C_2 = 20$$

$$C_2 = 10$$

$$C_1 \ = \ 1 \ ; \ C_2 \ = \ 10 \ ; \ C_2 \ = \ 2$$

Función de tiempo: $t(n) = n + 10 * log_2(n) * n + 2 \ \in \ \Theta(log_2(n) * n)$

$$t(n) = \begin{cases} n = 0 & entonces \quad 0 \\ n = 1 & entonces \quad 6 \\ n > 1 & entonces \quad n + 10 * log_2(n) * n + 2 \end{cases}$$

2.2.3 Observaciones y estimaciones

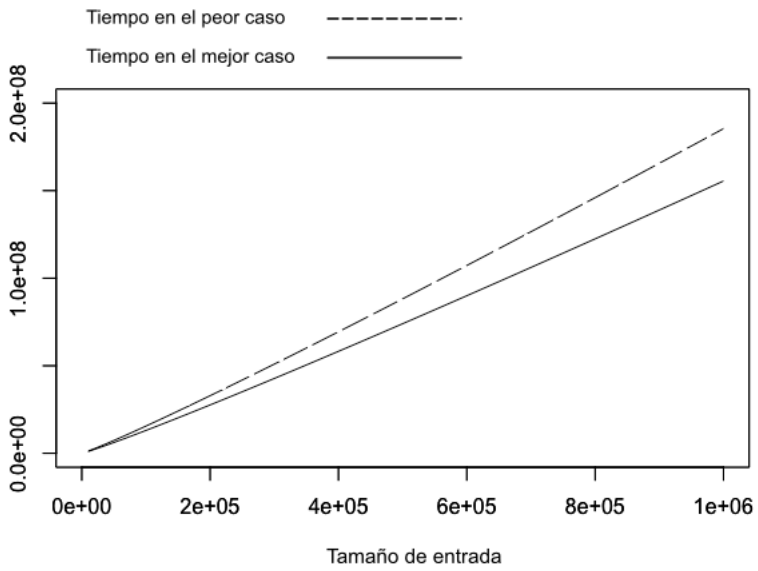


Figure 2.1: Caption

Tiempo (milis.)	**O(n)**	$\Omega(n)$	$\Theta(n)$
Tiempo en el mejor caso	$n * log_2(n)$	$(n * log_2(n)$	$n * log_2(n)$
Tiempo en el peor caso	$n * log_2(n)$	$n * log_2(n)$	$n * log_2(n)$
Tiempo promedio	$n * log_2(n)$	$n * log_2(n)$	$n * log_2(n)$

Tanto en el peor caso como en el mejor caso, ambos tienen la misma complejidad y se diferencian en las constantes. Por tanto, el tiempo promedio también tendrá una función de tiempo de orden $O(n * log_2(n)))$. La constante de su o-pequeña tendrá un valor entre 8,5 y 10.
El hecho de que el crecimiento sea de orden $\Theta(n \cdot \log_2(n))$ es debido al calculo de las subcadenas que tiene un orden $O(n)$. Si el calculo de una subcadena tuviera un coste constante a, entonces tiempo de ejecución sería lineal.

$$t(n) = 1 + 1 + 2 \cdot a + t(\tfrac{n}{2}) + t(\tfrac{n}{2}) + 1 + 1 + 3 + 1$$
$$t(n) = 7 + 2 \cdot a + 2t(\tfrac{n}{2})$$
$$n = 2^k$$
$$t_k - 2t_{k-1} = 7 + 2 \cdot a$$
$$t_k = C_1 \cdot 1^k + C_2 \cdot 2^k$$
$$k = log_2(n)$$
$$t(n) = C_1 \cdot n^{log_2 1} + C_2 \cdot n^{log_2 2}$$
$$t(n) = C_2 \cdot n \in O(n)$$

3 Casos de prueba y diseño de validación

3.1 Casos de prueba

Tras el diseño del algoritmo y el estudio teorico de sus tiempos de ejecución, se ha dispuesto a proceder a la implementación del algoritmo en el lenguaje C++.

Para la validación del algoritmo, se han utilizado como casos de prueba:

- cadena1 = "abcaeaabbcbekaabbcsabzzzyxaaaaa" y
  cadena2 = "abcaeookfjbekiebtcsabuvwyxaraaa".
  Resultado: La secuencia de vocales más larga tiene longitud 4 y
  comienza en la posición 3.

- cadena1 = "abcaeaabbcbekaabbcsabzzzyxaaaaa" y
  cadena2 = "abcaioubbcbekaabbcsabqwyxaaaaa". Resultado: La secuencia de vocales más larga

tiene longitud 5 y
comienza en la posición 26.

- cadena1 = "eeiioouuaaiiabcaeaabbcbekaabbcaeiouaesabzzzyxaaaaa" y
  cadena2 = "aeeiaoiuiaaiabcaeaabbcbekaabbcaeiouaesabzzzyxaaaaa". Resultado: La secuencia de vocales más larga tiene longitud 13 y comienza en la posición 0.

Consideramos que estas cadenas de prueba son suficientes para comprobar de manera visual que el algoritmo funciona.
Para cadenas más largas, se ha implementado un algoritmo iterativo en el que se devuelve un resultado y se compara con el resultado obtenido en el algoritmo de Divide y Vencerás para una misma cadena.

3.2 Diseño de validación

Para la validación del algoritmo se ha optado por implementar una función alternativa que utiliza un método iterativo. Se hará una comparación de resultados entre este algoritmo iterativo y el original de *Divide y vencerás* para comprobar que la implementación es correcta. Se ha creado la función `random_string(size_t longitud, int tipo)`. Esta función devuelve una cadena aleatoria cuya longitud se le pasa como parámetro. El parámetro `tipo` designa el tipo de cadena. Si el tipo es igual a cero, devuelve una cadena solo de consonantes. Si el tipo es igual a 1, devuelve una cadena solo de vocales. En otro caso, devuelve una cadena normal.

3.3 Generador de casos de prueba

Con la función `random_string(size_t longitud, int tipo)` generamos una cadena de tamaño n que se le pasa como parámetro. Dicha función puede generar arrays con solo vocales, solo consonantes o letras del alfabeto en general.
Se han utilizado las siguientes entradas para los casos de prueba: **27324, 500000, 1254721, 5468659, 15561223, 20000000, 40000000**.

3.4 Resultados de los casos de prueba

El caso más favorable se da cuando una cadena de tamaño n, siendo n un tamaño muy grande, **solo tenga CONSONANTES**. El caso más desfavorable se da cuando cuando una cadena **solo tiene**

VOCALES.

Tamaño de entrada: `27.324`

Tamaño de cadena: 27324	**Caso base 1**	**Caso base 2**	**Caso base 3**	**Caso base 4**
Tiempo en el mejor caso	11,803	7,758	6,265	5,259
Tiempo en el peor caso	41,036	31,771	26,383	24,475
Tiempo promedio	12,967	9,054	6,974	5,883

Tamaño de entrada `50.000`

Tiempo (milis.)	**Caso base 1**	**Caso base 2**	**Caso base 3**	**Caso base 4**
Tiempo en el mejor caso	21,547	15,625	10,011	9,735
Tiempo en el peor caso	77,648	68,340	49,089	48,168
Tiempo promedio	24,003	18,290	11,933	11,467

Tamaño de entrada: `1.254.721`

Tiempo (milis.)	**Caso base 1**	**Caso base 2**	**Caso base 3**	**Caso base 4**
Tiempo en el mejor caso	540,343	355,685	283,185	264,387
Tiempo en el peor caso	2202,08	2095,29	149,447	1494,47
Tiempo promedio	596,800	409,935	332,674	308,277

Tamaño de entrada: `5.468.659`

Tiempo (milis.)	**Caso base 1**	**Caso base 2**	**Caso base 3**	**Caso base 4**
Tiempo en el mejor caso	2269,2	1577,55	1142,22	1171,66
Tiempo en el peor caso	9912,68	8109,17	6972,56	7017,20
Tiempo promedio	2567,46	1808,47	1388,63	1375,14

Tamaño de entrada: `15.561.223`

Tiempo (milis.)	**Caso base 1**	**Caso base 2**	**Caso base 3**	**Caso base 4**
Tiempo en el mejor caso	5036,34	4407,6	3987,69	2833,39
Tiempo en el peor caso	29627,60	23448,1	28387,10	22055
Tiempo promedio	7943,48	5057,7	4596,6	3489,2

Tamaño de entrada: `20.000.000`

Tiempo (milis.)	**Caso base 1**	**Caso base 2**	**Caso base 3**	**Caso base 4**
Tiempo en el mejor caso	8346,71	5747,38	4803,7	4262,86
Tiempo en el peor caso	45555	30940,4	34932,5	33921,8
Tiempo promedio	10018,4	6588,87	5461,18	5091,47

Tamaño de entrada: `40.000.000`

Tiempo (milis.)	Caso base 1	Caso base 2	Caso base 3	Caso base 4
Tiempo en el mejor caso	16747,6	11725	9409,47	8866,54
Tiempo en el peor caso	79375,1	63916,9	72358,8	70603,6
Tiempo promedio	20148,8	13800,5	11245	10414,1

Dados los resultados, se puede observar que teniendo un caso base igual a 4, se obtiene un tiempo de ejecución menor. Por lo que nos decantamos por utilizar un tamaño cadena igual a 4 para el caso base.

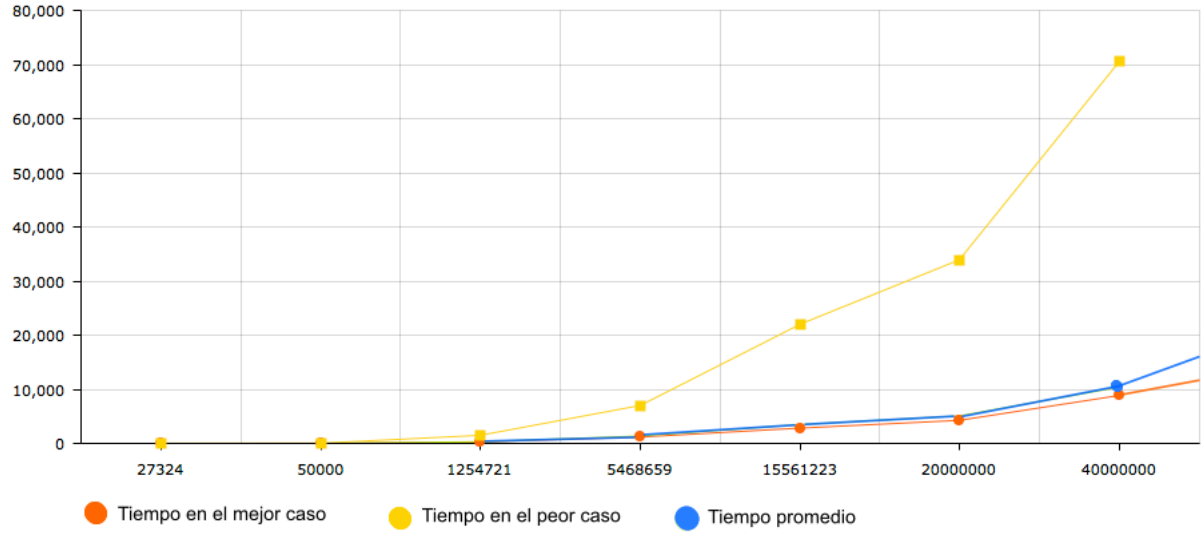


Figure 3.1: Gráfica de resultados

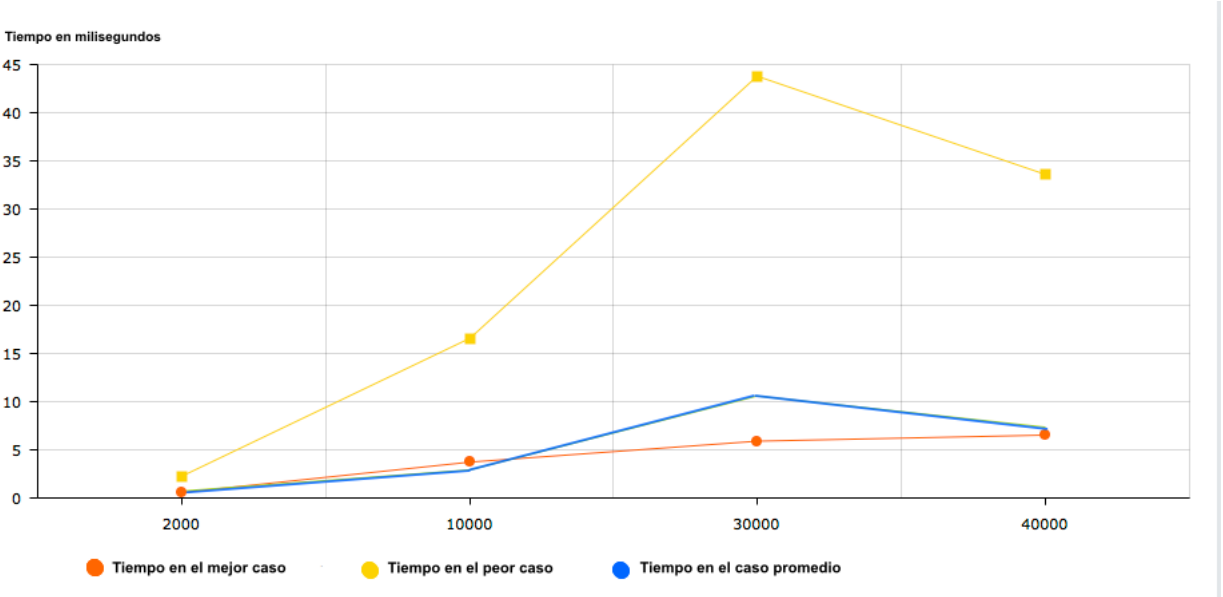


Figure 3.2: Gráfica de resultados ampliada

3.4.1 Comparativa de resultados

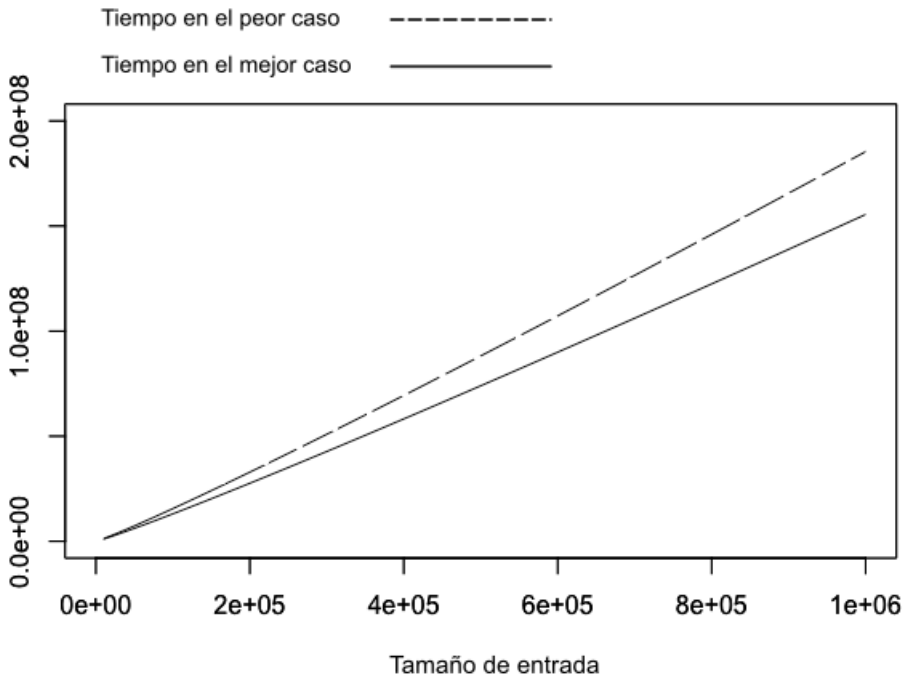


Figure 3.3: Gráfica del estudio teórico

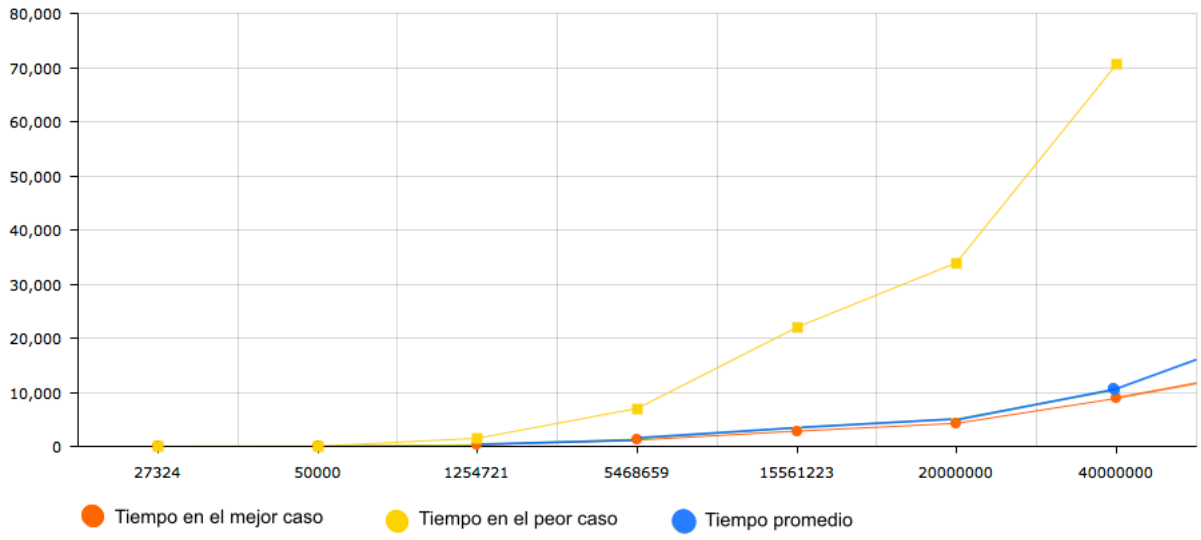


Figure 3.4: Gráfica de los resultados empíricos

Observando las gráficas del estudio teórico y resultados empíricos podemos comprobar que existe una enorme discrepancia en los resultados.

1. El crecimiento en la gráfica de resultados es mucho más lento que en el estudio teórico.
2. Los resultados del tiempo en peor caso son mucho mayores. En el estudio teórico, la diferencia entre el tiempo más favorable y más desfavorable es muy baja, mientras que en la gráfica de

resultado es bastante más pronunciada, sobretodo cuando el tamaño de las cadenas son mayores.

3. El crecimiento del tiempo promedio dista muy poco del tiempo más favorable.

4. El crecimiento del tiempo más desfavorable tiene un crecimiento que, conforme es más grande el tamaño de la cadena, se asemeja a una función parabólica.

Sin embargo, existen varias justificaciones al respecto de esta diferencia. Dentro de la sección de combinación, las funciones de frontera `get_parte_mas_derecha()` y `get_parte_mas_izquierda()` consumen más tiempo del estimado en el estudio teórico. Haciendo una prueba y eliminando estas dos funciones, el resultado para el tiempo en el caso más desfavorable es:

Tiempo en el peor caso

Tamaño de cadena	Con las funciones de frontera	Sin funciones de frontera
27.324	24,475	7,151
50.000	48,168	14,02
1.254.721	1494,47	377,37
5.468.659	7017,29	1950,2
15.561.223	22055	4129,32
20.000.000	33921,8	7263
40.000.000	70603,6	14107,2

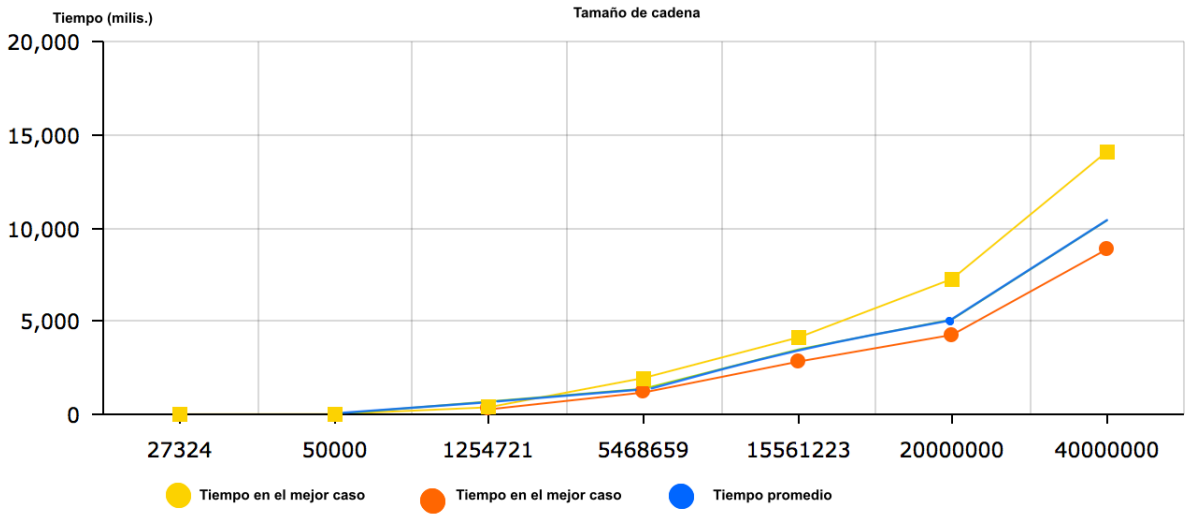


Figure 3.5: Gráfico con el $t_M(n)$ sin funciones de frontera

Sin las funciones de la frontera, podemos ver que el tiempo de ejecución en el peor caso crece de manera similiar al tiempo en el mejor caso y tiempo promedio, y la diferencia de resultados de los tiempos es menor.

Otra justificación con respecto al tiempo promedio y al tiempo en el peor caso, es que la probabilidad de que una letra sea vocal es de $\frac{5}{26}$, mientras que la probabilidad de que sea consonante es de $\frac{21}{26}$, por

lo que es lógico que exista una mayor probabilidad que resulte ser consonante. Por tanto, es lógico que la línea de crecimiento del tiempo promedio sea muy similar al del tiempo más favorable.

4 Conclusiones y observaciones

Tras comparar los datos del estudio teórico con los resultados empíricos de los casos de prueba, hemos constatado que existe una similitud en el crecimiento de ambos estudios. La variación del crecimiento de ambos crecimientos suele ser debido a varios factores de dependencia de funciones y el tipo de lenguaje utilizado. En cuanto al tiempo promedio, su variación puede depender del número de vocales consecutivas que tenga una cadena.

Con este proyecto, he podido experimentar y llevar a la práctica lo que he aprendido sobre el cálculo de la complejidad y el conteo de instrucciones. Es fundamental poder prever el crecimiento de una función así como tener una idea de los recursos necesarios para que ejecutar un algoritmo de manera correcta. Me ha parecido satisfactorio poner en práctica los conocimiento adquiridos de conteo, ecuación característica y estimación del tiempo y poder contrastarlo con los resultados reales obtenidos en los casos de prueba. El tiempo estimado invertido en la práctica es el siguiente desglose:

Tareas	**Tiempo estimado (minutos)**
Diseño del algoritmo	250 min
Estudio teorico	90 min
Implementación en C++	200 min
Solución de errores del programa en C++	500 min
Validación del algoritmo con pruebas	150 min
Implementación de generador de caso	30 min
Recopilación de resultados de prueba	60 min
Documentación	300 min
TOTAL	**1580 min (26 h.)**