# MEMORIA DE LA PRÁCTICA 1

## Divide y Vencerás


Algoritmos y Estructuras de Datos II

Pablo Belchí Corredor

Daniel Cascales Marcos

G1.2

# Índice

1. Introducción
2. Pseudocódigo y explicación del algoritmo
3. Explicación del código
   3.1.
4.
5.

# Introducción

Este trabajo consiste en resolver un problema a través del método divide y vencerás para posteriormente hacer un estudio de los resultados y de la eficiencia del código.
El método divide y vencerás consiste en identificar el problema y dividirlo de forma recursiva hasta llegar a un subproblema mucho más simple para tratarlo, después se combinan todos los resultados de los subproblemas y se obtiene la respuesta al problema inicial.

Enunciado del problema :

*2) Dadas dos cadenas A y B de la misma longitud n, y un natural m  n, encontrar C, la subcadena de A 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 incluidas en C). 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.*

*Ejemplo: n=10, m=5*

*A= c d d a b c d a c c*
*B= c a c d d b c a d c*
*dife: 0 3 1 3 2 1 1 0 1 0*

*Solución: C, posición de inicio igual a 2, y diferencia total igual a 10*

# 2.PSEUDOCÓDIGO Y EXPLICACIÓN DEL ALGORITMO

```
//Funcion para calcular la diferencia entre las dos cadenas
FUNCION calcularDiferencia(A, B):
    diferencias <- VECTOR DE TAMAÑO A.size()
    PARA i <- 0 HASTA A.size() - 1 HACER
        diferencias[i] <- ABS(A[i] - B[i])
    FIN PARA
    RETORNAR diferencias
FIN FUNCION


//Funcion para combinar las soluciones encontradas
FUNCION combinar(izquierda, derecha, diferencias, s, t, h, m):
    mejorInicio <- -1
    maximaSuma <- -1

    PARA i <- s HASTA h HACER
        fin <- MIN(i + m - 1, t)
        SI (fin - i + 1 == m) ENTONCES
                            sumaActual  <-  SUMA(diferencias[i]  HASTA
diferencias[fin])
            SI sumaActual > maximaSuma ENTONCES
                maximaSuma <- sumaActual
                mejorInicio <- i
            FIN SI
        FIN SI
    FIN PARA

    PARA i <- h + 1 HASTA t HACER
        inicio <- MAX(i - m + 1, s)
        SI (i - inicio + 1 == m) ENTONCES
                        sumaActual  <-  SUMA(diferencias[inicio]  HASTA
diferencias[i])
            SI sumaActual > maximaSuma ENTONCES
                maximaSuma <- sumaActual
                mejorInicio <- inicio
            FIN SI
        FIN SI
    FIN PARA

    RETORNAR (mejorInicio, maximaSuma)
FIN FUNCION
```

```
//Funcion para tratar los casos base y dividir recursivamente el problema
FUNCION divideYvenceras(diferencias, s, t, m):
    SI (s == t Y m == 1) ENTONCES
        RETORNAR (s, diferencias[s])
    FIN SI

    SI ((t - s + 1) < m) ENTONCES
        RETORNAR (-1, -1)
    FIN SI

    SI ((t - s + 1) == m) ENTONCES
        diferenciaTotal <- SUMA(diferencias[s] HASTA diferencias[s + m - 1])
        RETORNAR (s, diferenciaTotal)
    FIN SI

    h <- (s + t) / 2
    izquierda <- divideYvenceras(diferencias, s, h, m)
    derecha <- divideYvenceras(diferencias, h + 1, t, m)
    combinado <- combinar(izquierda, derecha, diferencias, s, t, h, m)

    SI izquierda.segundo >= derecha.segundo Y izquierda.segundo >= combinado.segundo ENTONCES
        RETORNAR izquierda
    SI NO SI derecha.segundo >= izquierda.segundo Y derecha.segundo >= combinado.segundo ENTONCES
        RETORNAR derecha
    FIN SI
    RETORNAR combinado
FIN FUNCION


FUNCION encontrarMaximaDiferenciaSubcadena(A, B, m):
    diferencias <- calcularDiferencia(A, B)
    RETORNAR divideYvenceras(diferencias, 0, diferencias.size() - 1, m)
FIN FUNCION

PROCEDIMIENTO PRINCIPAL
    SI argumentos < 2 ENTONCES
        IMPRIMIR "Uso: programa archivo_entrada.txt"
        TERMINAR
    FIN SI
```

```
    archivo <- ABRIR archivo_entrada.txt
    SI archivo NO EXISTE ENTONCES
        IMPRIMIR "No se pudo abrir el archivo."
        TERMINAR
    FIN SI

    LEER strA DESDE archivo
    LEER strB DESDE archivo
    LEER m DESDE archivo
    CERRAR archivo

    inicioTiempo <- OBTENER_TIEMPO()

    strA <- REMOVER_ESPACIOS(strA)
    strB <- REMOVER_ESPACIOS(strB)

    A <- CONVERTIR_A_VECTOR_ASCII(strA)
    B <- CONVERTIR_A_VECTOR_ASCII(strB)

    resultado <- encontrarMaximaDiferenciaSubcadena(A, B, m)

    finTiempo <- OBTENER_TIEMPO()
    duracion <- finTiempo - inicioTiempo

        IMPRIMIR "Posición de inicio: ", resultado.primero, "
Diferencia total: ", resultado.segundo
    IMPRIMIR "Tiempo de ejecución: ", duracion, " microsegundos"
FIN PROCEDIMIENTO PRINCIPAL


FIN
```

# 3.EXPLICACIÓN DEL CÓDIGO

## *3.1 MÉTODO CalcularDiferencia*

*-Recibe dos vectores enteros "A" y "B".*
*-De estos dos vectores calcula la diferencia absoluta entre sus elementos.*
*-Devuelve el vector de las diferencias.*

```
vector<int> calcularDiferencia(const vector<int> &A, const vector<int> &B) {
    vector<int> diferencias(A.size());
    for (int i = 0; i < A.size(); ++i) {
        diferencias[i] = abs(A[i] - B[i]);
    }

    return diferencias;
}
```

## *3.2 MÉTODO combinar*

*-Recorre la mitad izquierda para verificar si extendiéndose a la derecha puede formarse una mejor subcadena tamaño m.*
*-Recorre la mitad derecha para verificar si extendiendola a la izquierda puede formarse una mejor subcadena.*
*-Devuelve la posición de inicio y diferencia máxima.*

```
pair<int, int> combinar(const pair<int, int> &izquierda, const pair<int, int> &derecha,
                        const vector<int> &diferencias, int s, int t, int h, int m) {
    int mejorInicio = -1;
    int maximaSuma = -1;

    // Buscar en la mitad izquierda extendiéndose a la derecha si es necesario
    for (int i = s; i <= h; ++i) {
        int fin = min(i + m - 1, t); // Limitar el extremo derecho al límite del array
        if (fin - i + 1 == m) { // Asegurarse de que tenga tamaño m
            int sumaActual = accumulate(diferencias.begin() + i, diferencias.begin() + fin + 1, 0);
            if (sumaActual > maximaSuma) {
                maximaSuma = sumaActual;
                mejorInicio = i;
            }
        }
    }
```

```
    // Buscar en la mitad derecha extendiéndose a la izquierda si es necesario
    for (int i = h + 1; i <= t; ++i) {
        int inicio = max(i - m + 1, s); // Limitar el extremo izquierdo al límite del array
        if (i - inicio + 1 == m) { // Asegurarse de que tenga tamaño m
            int sumaActual = accumulate(diferencias.begin() + inicio, diferencias.begin() + i + 1, 0);
            if (sumaActual > maximaSuma) {
                maximaSuma = sumaActual;
                mejorInicio = inicio;
            }
        }
    }

    // Devolver el mejor resultado encontrado
    return {mejorInicio, maximaSuma};
}
```

## *3.3 MÉTODO divideYvenceras*

*-Implementa el algoritmo Divide y Vencerás para que encontremos la subcadena m con la mayor suma de diferencias.*
*-En el caso base si "s==t" y "m==1" devuelve la única diferencia posible. Si la subcadena tiene menos de m elementos, devuelve "{-1,-1}". Y por último si tiene m elementos devuelve la suma total de diferencias.*
*-En el caso recursivo divide el problema en dos mitades. Encuentra la mejor solución en la izquierda y derecha. Por último usa combinar y nos devuelve la mejor opción entre izquierda, derecha y combinada.*

```
pair<int, int> divideYvenceras(const vector<int> &diferencias, int s, int t, int m) {
    if (s == t && m == 1) {
        return {s, diferencias[s]};
    }
    if ((t - s + 1) < m) {
        return {-1, -1};
    }
    if ((t - s + 1) == m) {
        int diferenciaTotal = accumulate(diferencias.begin() + s , diferencias.begin() + s + m, 0);
        return {s,diferenciaTotal};
    }

    int h = (s + t) / 2;
    pair<int, int> izquierda = divideYvenceras(diferencias, s, h, m);
    pair<int, int> derecha = divideYvenceras(diferencias, h + 1, t, m);
    //return combinar(izquierda, derecha, diferencias, s, t, h, m);
    pair<int, int> combinado = combinar(izquierda, derecha, diferencias, s, t, h, m);


    //Devolver la mejor opción entre izquierda, derecha y combinada
    if (izquierda.second >= derecha.second && izquierda.second >= combinado.second) return izquierda;
    if (derecha.second >= izquierda.second && derecha.second >= combinado.second) return derecha;
    return combinado;
}
```

## *3.4 MÉTODO encontrarMaximaDiferenciaSubcadena*

*-Calcula la diferencia absoluta entre A y B.*
*-Llama a divide y vencerás para resolver el problema.*

```
pair<int, int> encontrarMaximaDiferenciaSubcadena(const vector<int> &A, const vector<int> &B, int m) {
    vector<int> diferencias = calcularDiferencia(A, B);
    return divideYvenceras(diferencias, 0, diferencias.size() - 1, m);
}
```

## *3.5 MÉTODO main*

*-Verifica el archivo de entrada. Abre y extrae A, B y m.*
*-Elimina los espacios en strA y strB.*
*-Convierte los Strings a vectores de enteros.*
*-Llama a encontrarMaximaDiferenciaSubcadena y mide el tiempo de ejecución.*
*-Imprime la posición inicial y la diferencia total.*
*-Muestra el tiempo de ejecución en microsegundos.*

```
int main(int argc, char* argv[]) {
    if (argc < 2) {
        cout << "Uso: " << argv[0] << " archivo_entrada.txt" << endl;
        return 1;
    }

    ifstream archivo(argv[1]);
    if (!archivo) {
        cout << "No se pudo abrir el archivo." << endl;
        return 1;
    }

    string strA, strB;
    int m;

    // Leer datos desde el archivo
    getline(archivo, strA);
    getline(archivo, strB);
    archivo >> m;

    archivo.close();

    auto start = chrono::high_resolution_clock::now();

    // Eliminar espacios de las cadenas
    strA.erase(remove(strA.begin(), strA.end(), ' '), strA.end());
    strB.erase(remove(strB.begin(), strB.end(), ' '), strB.end());

    vector<int> A(strA.begin(), strA.end());
    vector<int> B(strB.begin(), strB.end());

    pair<int, int> resultado = encontrarMaximaDiferenciaSubcadena(A, B, m);

    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start).count();

    cout << "Posición de inicio: " << resultado.first << ", Diferencia total: " << resultado.second << endl;
    cout << "Tiempo de ejecución: " << duration << " microsegundos" << endl;

    return 0;
}
```

# 4. Estudio Teórico

## 4.1 Tiempo mejor caso

El mejor caso en este programa es cuando al tamaño de la subcadena que buscamos y es el mismo que el de las cadenas de caracteres ya que el algoritmo solo tendría que devolver la suma de todas las diferencias. Este caso tendría un orden de O(n).

## 4.2 Tiempo peor caso

En este problema el peor caso es el mismo que el caso promedio ya que da igual lo que contengan las cadenas porque el array siempre se recorre entero buscando en todas las posiciones. La función más costosa es combinar() y el programa en estos casos tiene un orden de O(n log n).

## 4.3 Orden de ejecución

La recurrencia que describe el tiempo de ejecución es:

**T( n ) = 2T(n/2) + O(n)**

donde:

- **2T(n/2)** proviene de las dos llamadas recursivas a divideYvenceras().
- **O(n)** es el tiempo que tarda combinar(), ya que recorre hasta nnn elementos en el peor caso.

Aplicando el Teorema Maestro a la ecuación de recurrencia:

- **a =2** (dos subproblemas)
- **b = 2** (cada subproblema tiene tamaño n/2)
- **f(n) = O(n)** (tiempo de combinación)

El caso relevante es f(n)=O(n), que corresponde a la forma O(n^c) con c=1, y como c=log_b a, es decir, 1=log_2 2 = 1, la complejidad final es:

**T(n) = O(n logn)**

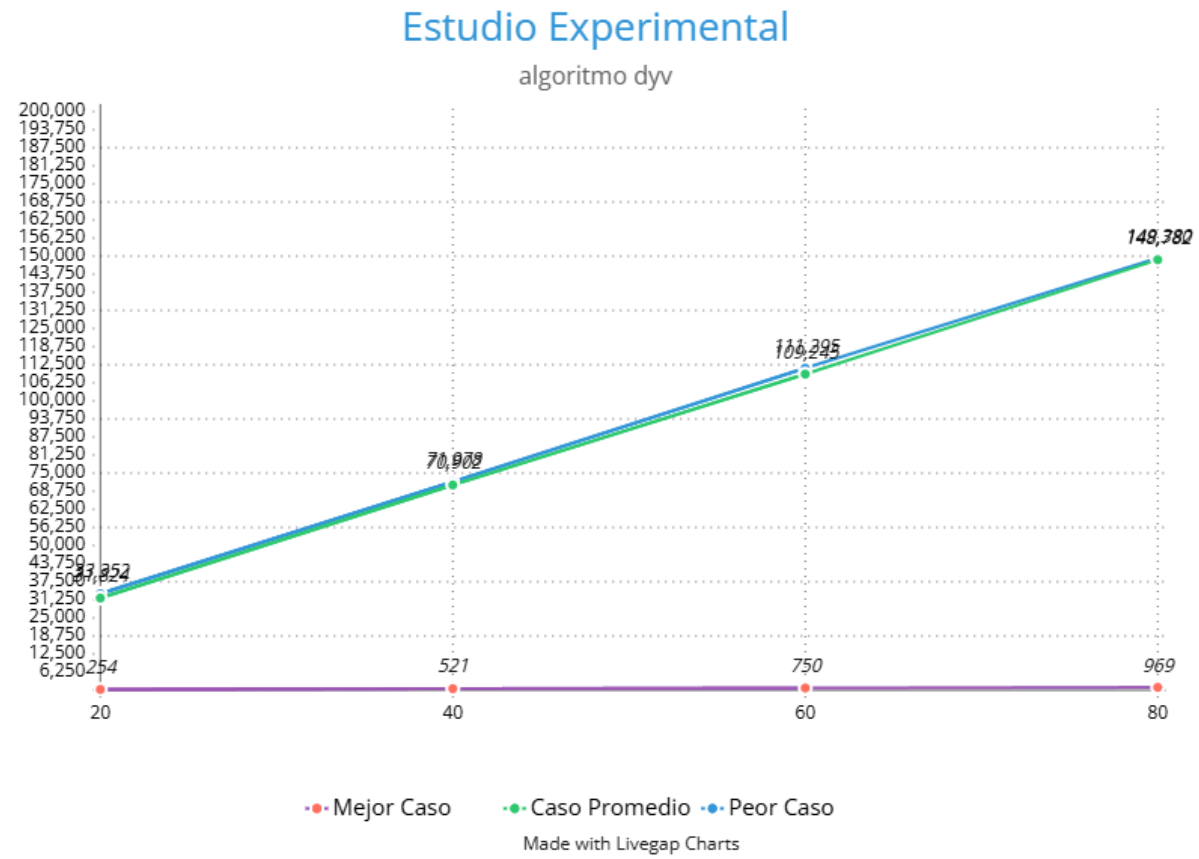
# 5. Validación del Algoritmo

Para la validación del algoritmo hemos creado otro programa para resolver el mismo problema pero enfrentándolo sin utilizar el método de divide y vencerás. Con este programa obtenemos las soluciones directamente y las comparamos con las soluciones conseguidas con el primer programa para validarlas. Al introducir las mismas entradas hemos podido comprobar que efectivamente las salidas coinciden y por la tanto el método está bien implementado en la resolución del problema.

# 6. Estudio Experimental

Hemos realizado pruebas con distintos tamaños. Tanto los casos más favorables para el algoritmo como algunos más extremos. Hemos utilizado diferentes entradas para validar los casos mejor, peor y promedio. El mejor caso es cuando m es del mismo tamaño de la cadena. El peor caso del promedio tienen una diferencia ínfima, ya que dependen de las cadenas que les introduzcas.

# 7. Contraste del Estudio Teórico y Experimental




A la conclusión que hemos llegado es que O(n) va a representar la cota inferior del tiempo de ejecución del algoritmo, y O(n log n) la cota superior, Esto nos permitirá saber cuales son los recursos máximos que necesitaremos en nuestro algoritmo. Además en esta gráfica podemos ver como va incrementando el tiempo de ejecución en función del tamaño de la entrada.

| Tamaño(Kb) | Mejor Caso (microsegundos) | Caso Promedio (microsegundos) | Peor Caso (microsegundos) |
|---|---|---|---|
| 20 | 254 | 31824 | 33352 |
| 40 | 521 | 70902 | 71978 |
| 60 | 750 | 109245 | 111295 |
| 80 | 969 | 148782 | 149380 |

# 8. Conclusiones

En este trabajo, hemos abordado la resolución de un problema de programación aplicando el método de Divide y Vencerás. A través de la descomposición del problema en subproblemas más pequeños, logramos simplificar su solución y optimizar el rendimiento del algoritmo.

Lo que más nos costó acerca de elaborar una solución con esta estrategia fue el implementar como buscar las subcadenas de tamaño m entre las fronteras ya que se tienen que comprobar todas las fronteras que se generan al dividir el problema y buscar digito a digito la subcadena con la suma de diferencias más grande.

En resumen, Divide y Vencerás es una técnica muy útil en programación, ya que permite resolver problemas de manera más eficiente y estructurada.