Práctica 3
Algoritmos Voraces y Programación Dinámica

Alumno: Jorge Valenzuela García

Suma hasta un número
VORAZ

1. Datos del problema:

S: Conjunto de candidatos. Son el conjunto de números enteros que hay dentro del archivo .txt que pasemos como parámetro al programa. Representados en el código con la variable *secuencia*.

M: Es el valor que deben sumar. Representado en el programa por la variable *suma*.

Solución: Es una subsecuencia de S. Son los valores que sumados dan como resultado M.

2. Representación de la solución:

Una Solución = $(x_1, x_2, \dots, x_n)$ siendo $x_i = 0$ ó $x_i = 1$ donde x_i representa si se coge o no el elemento de la secuencia.

- Si $x_i = 1$, se coge el elemento de la secuencia S.
- Si $x_i = 0$, no se coge el elemento.

3. Formulación matemática:

Maximizar $\sum_{i=1}^{n} \mathbf{x_i \cdot S(i)}$ Siendo S(i) el elemento en cuestión de la secuencia.
sujeto a la restricción
$\sum_{i=1}^{n} \mathbf{x_i \cdot S(i) \leq M}$ **y**
$\mathbf{x_i = 0}$ **ó** $\mathbf{x_i = 1}$

4. Diseño de la solución:

- Candidatos:

Es S, el conjunto de enteros que podemos sumar.

- Función Solución:

Esta función en este ejercicio no está definida en sí misma. Esto es debido a que dentro de la función selección ya comprobamos si la subsecuencia encontrada es ya una solución.
Para saber si es solución, los elementos elegidos durante la selección y que han sido añadidos al vector llamado *subsecuencia* deben de sumar el valor M.

- Función de Selección

Escogemos el valor más prometedor del conjunto S. En este caso el valor más prometedor es el mayor de todos los valores de S pero que no sobrepase el valor M. Una vez hecho esto comprobamos si el valor escogido es factible (ver siguiente apartado).

- Función Factible

Esta función también está implícita dentro de la función de selección. Comprobamos si el candidato seleccionado no sobrepasa la suma buscada, además de cumplir que este candidato, junto con la suma de los valores ya seleccionados previamente, no supere M.
Por tanto, se puede entender que este método será true si el candidato seleccionado no es mayor que M y no sobrepasa a M al ser sumado junto al resto de candidatos previamente escogidos. Si es factible ver siguiente apartado. Si no es así, volvemos a la función de selección.

- Añadir un objeto a la solución

Añadimos el valor si es factible.

- Función Objetivo

Suma los valores de los candidatos escogidos que den como resultado M.

La función de selección no siempre encuentra el valor óptimo. Veamos el siguiente caso:
S = 100, 75, 41 y 39
M = 80
En este caso, el valor escogido por la función de selección será el 75, y al ser factible lo añadirá al conjunto de candidatos solución. Este conjunto al solo tener el 75 no es la solución aún, ya que necesitamos otro u otros valores más, que al sumarlos con 75, nos de 80, por lo que la función de selección buscará un nuevo candidato para ello, pero vemos que no lo va a encontrar, debido a que 75 + 41 o 75 + 39, ambos superan 80, por lo que la solución final para este caso será 75, pese a que a simple vista vemos que 41+39 = 80 (solución óptima).

• Código

El código es el siguiente:

```
/**     @brief: Método que selecciona de un vector de enteros un subconjunto de ellos, cuyo
                resultado es igual al parámetro suma.
        @param secuencia: Conjunto de enteros.
        @param suma: valor entero que busca ser el resultado de la suma del subconjunto elegido
                     del conjunto secuencia.
        @param subsecuencia: es el subconjunto resultado, cuya suma da como resultado el
                             parámetro suma.
*/
void seleccion(const vector<int> &secuencia, int suma, vector<int> & subsecuencia){
        vector <int> resultado;
        vector <int> usada = secuencia;
        cerr << "Llamado metodo seleccionar" << endl;

        int n = secuencia.size();       // Numero de elementos de la secuencia
        int mayor;                      // Sera el numero mayor que no sobrepase la suma. Buscamos
                                        // asi la optimizacion
        int acumulacion = 0;            // Es el valor entero que va aculumando la suma de los valores
                                        // enteros que vayamos eligiendo
        int posicion;                   // Sera la posicion donde este el mayor valor, es decir, el valor
                                        // que pase a acumularse

        while ((acumulacion < suma   or
                acumulacion != suma) and
                usada.size() > 0){      // Mientras la acumulacion de valores (suma) sea menor que el
                                        // resultado o distinto del mismo...
                        mayor = -1;     // Antes de recorrer el vector pasamos a mayor -1, se supone
                                        // que siempre tratamos con enteros positivos
                        for (int i = 0; i < n; i++){        // Creamos este bucle para encontrar el mayor
                                                            // elemento de secuencia que no sobrepase a
                                                            // suma
                                if (mayor < usada[i] and usada[i] <= suma and acumulacion+usada[i]
<= suma){
                                        mayor = usada[i];
                                        posicion = i;
                                }
                        }
                        if (mayor != -1){       // Si es distinto de menos 1 quiere decir que ha
                                                // encontrado un nuevo valor para la solucion
                                acumulacion = mayor + acumulacion;      // Guardamos en
                                                                        // acumulacion la suma de
                                                                        // la anterior acumulacion
                                                                        // con mayor
                                subsecuencia.push_back(usada[posicion]); // Añadimos a la
                                                                         // subsecuencia el valor
                                                                         // encontrado
                                usada.erase(usada.begin()+posicion);    // Borramos ese valor de
                                                                        // usada ya que lo hemos
                                                                        // contado ya para la suma
```

```
                        n--;      // Para actualizar la informacion del bucle
                    }
                    else{                   // Si esto ocurre es porque no se encuentran mas valores
                                            // para la solucion y ya estamos en el caso mas cercano
                                            // a la solucion
                        acumulacion = suma+1;       // Hacemos esto para que salga del while
                        cerr << "No se ha encontrado solucion" << endl;
                    }
        }
}
```

5. Eficiencia teórica:

La eficiencia del algoritmo completo vemos que es **O(n^2)**, esto se explica de la siguiente manera.

Vamos a dividirlo por partes ya que mi solución contiene todas las funciones implícitas en una sola.

Si nos centramos en la función de selección, vemos que siempre va a buscar el mayor valor del vector bajo una condición, lo cual va a recorrer siempre todo el vector, ya sea el mejor o el peor de los casos (es una búsqueda del mayor elemento en un vector, siempre es O(n) porque tenemos que recorrer todo el vector). Por lo que la función de selección, que es la que más nos interesa, tiene un orden de eficiencia de **O(n)** en el mejor y en el peor de los casos.

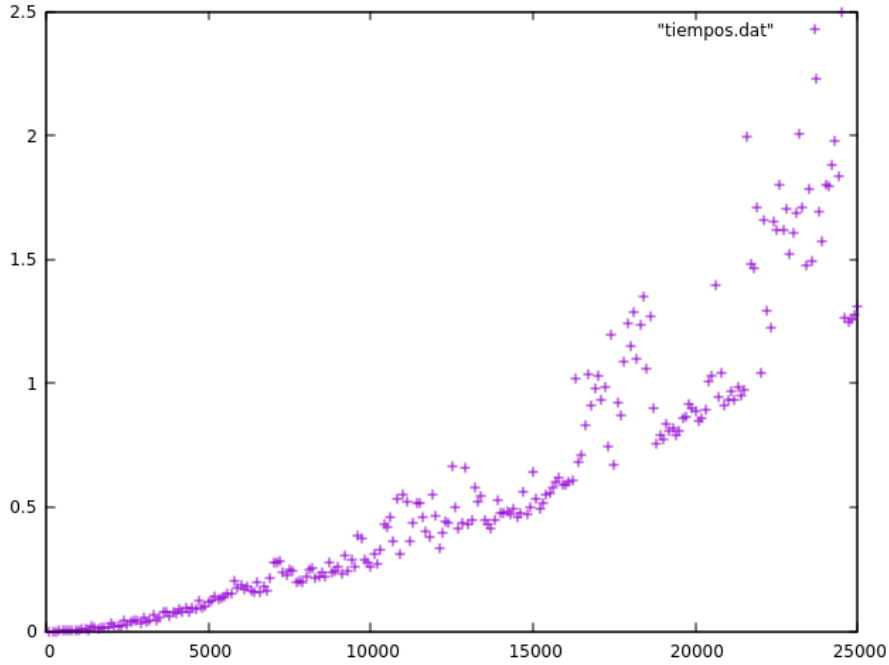
Ahora bien, si nos centramos en todo el método que he implementado, vemos que hasta encontrar la solución, vamos a tener que buscar varios candidatos, de hecho, en el peor de los casos vamos a tener que buscar a todos los candidatos, por lo que las iteraciones son n*n, es decir, el orden de eficiencia es de **O(n^2)**.
Para aclarar esto más aún, vemos que una vez encontramos un candidato lo borramos del vector a buscar, lo cual hace que el vector decrezca, si esto lo aplicamos a todos los elementos del vector, el número de iteraciones es de n*(n/2). Lo que finalmente vemos es que nos enfrentamos a un orden de **O(n^2)** en el peor de los casos.
En el mejor de los casos encontraríamos la solución con el primer candidato, lo cual nos deja un orden de eficiencia **Ω(n)**, que es igual al orden de eficiencia del método de selección, que es lo que se tarda en encontrar al candidato factible dentro de nuestro vector.

6. Eficiencia Empírica:

Para la eficiencia empírica he realizado un nuevo código en el que, en vez usar el archivo conjunto.txt, he creado un vector de tamaño creciente con números aleatorios entre 0 y 100, que hará de secuencia de elementos, mientras que el número que nos interesa como resultado de la suma siempre va a valer numeric_limits<int>::max(). Esto es así para que, pese al tamaño del vector, siempre forcemos lo máximo posible a obtener los resultados en el peor de los casos.
Conociendo el código del programa seguramente cuanto mayor sea el vector mejores resultados obtendremos. Esto es debido a que tendremos más elementos y el programa podrá llegar a la solución sin tener que usar todos los elementos del vector. Por ello intentamos independientemente del tamaño del vector, que los resultados obtenidos sean los más próximos al peor de los casos.

Tras obtener los resultados, los tiempos según el tamaño son los siguientes:

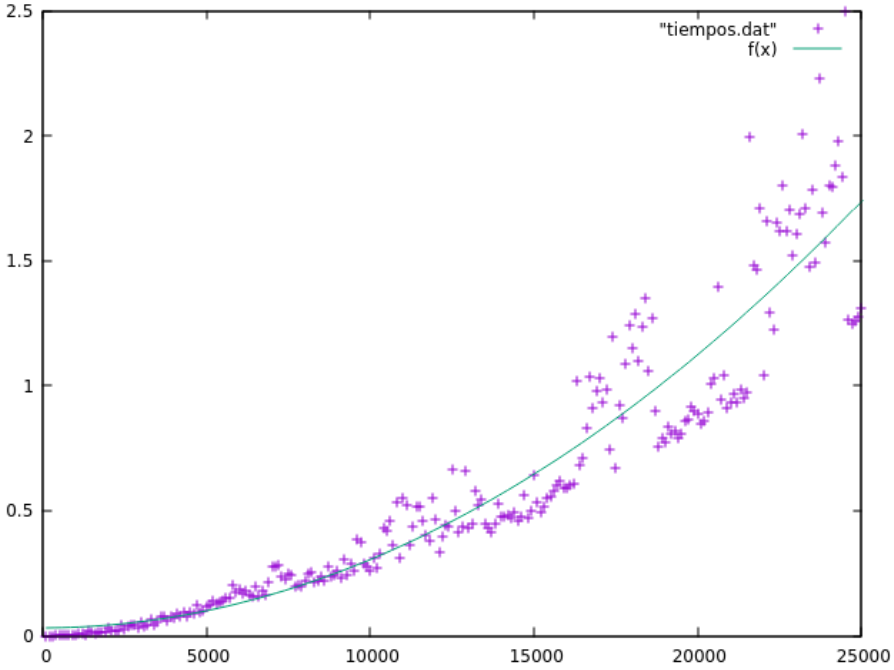
Vemos la función obtenida tiene forma cuadrática. Si no forzásemos el peor de los casos, casi de forma segura obtendríamos un función completamente lineal.



Los ajustes son los siguientes:

a = 2.73263e-09
b = 0.0315943

Finalmente si nos ajustamos a una curva de $O(n^2)$ vemos:



7. Dificultad:

La dificultad de este ejercicio ha sido de 4. Codificar el algoritmo voraz ha sido muy sencillo, pero el describir y contestar a cada uno de los puntos ha sido algo más complicado, debido a que es el primer ejercicio.

Suma hasta un número
PROGRAMACIÓN DINÁMICA

Lo primero que tenemos que ver es si se cumple el principio de optimalidad de Bellman. Para ello hay que comprobar si los subproblemas que tiene el problema en sí podemos resolverlos de forma óptima.
En este caso en concreto sí se cumple, ya que para cualquier M vamos a obtener una secuencia de valores resultado óptima.

De la siguiente forma para un problema X teniendo como valor a encontrar M obtendremos la secuencia (x_1, x_2, x_3 , …, x_n).
De modo que para un valor P + P_{i+1} la solución será la secuencia
(x_1, x_2, x_3 , …, x_n) + (x_{i+1}, x_{i+2}, … , x_{i+n}).
De esta forma siempre encontremos solución a los subproblemas.

Cabe decir que pese a que no encuentre la solución final, siempre encuentra la solución más próxima por lo que siempre obtenemos el mejor resultado para una secuencia X y un valor buscado M.

1. Ecuación recurrente del problema:

Tenemos dos casos:

1. Donde no cogemos el elemento, por tanto es un valor menos que tenemos que tener en cuenta.
2. Donde cogemos el elemento, por tanto, al igual que antes, tenemos un valor menos que tener en cuenta, además de añadir ese elemento al resultado.

- **M** representa el valor buscado, es decir, la suma de los elementos escogidos tiene que dar como resultado M.
- **k** es el elemento en cuestión. Representa un número entero positivo.

$$\text{Suma}(k, M) = \begin{cases} \max(\text{Suma}(k-1, M), \text{Suma}(k-1, M - \text{Valor}_k + 1)) \\ \text{No se coge} \qquad \text{Se coge el elemento} \end{cases}$$

2. Definir la estrategia de aplicación de la fórmula:

Se define una tabla con *n* filas y *m* columnas.
Siendo las filas los valores de los elementos de la secuencia aportada (k), y siendo las columnas la secuencia de números hasta llegar a M+1, es decir, si M fuese 10, las columnas serían (0, 1, 2, 3, …, 9, 10).

Una vez entendidos los campos que tiene tabla pasamos a rellenar la tabla. Para ello iremos completando la tabla quedándonos con el mínimo entre el valor i-1 y el valor de j-k +1, siendo *i* las filas y *j* las columnas. De este modo la tabla siempre tendrá el mínimo para cada par i,j, y ya solo nos quedará obtener la solución.

De esta forma completamos la tabla para todos los elementos k,M , además a la hora de codificar, los elementos deben estar ordenados de menor a mayor para el correcto funcionamiento del método.

3. Especificar como se reconstruye la solución final a partir de los valores de las tablas.

Una vez construida la tabla pasamos a crear la solución. Para ello se parte de la última casilla de todas, y vamos comprobando si el valor de la casilla actual es igual a la superior, de ser así sabemos que ese valor no hay que cogerlo por lo que ascendemos a la fila de arriba. Volvemos a comprobar si el valor de arriba al actual es diferente de éste, supongamos ahora que sí es diferente, por tanto, eso nos indica que ese valor hay que cogerlo.

Mi solución es bastante parecida a la que se ofrece en el problema del cambio de moneda, la única y gran diferencia es que en el cambio de moneda cuando encontramos una solución podemos seguir explorando esa fila para ver si hay otra solución, en cambio en mi problema eso no es factible, ya que solo podemos coger un elemento una vez, por lo que en cuanto encontramos una solución ascendemos a la fila de arriba.

4. Código:

```
void seleccion(const vector<int> &secuencia, int suma, vector<int> & subsecuencia){

	vector<int> usados = secuencia;
	int n = secuencia.size();

// Lo primero es ordenar los valores de la secuencia
	int aux, posicion;
	int menor = numeric_limits<int>::max();

	for (int i = 0; i < n; i++){
		for (int j = i; j < n; j++){
			if (menor > usados[j]){
				menor = usados[j];
				posicion = j;
			}
		}
		aux = usados[i];
		usados[i] = menor;
		usados[posicion] = aux;
		menor = numeric_limits<int>::max();
	}

// Pasamos a crear la tabla

	int tabla[n][suma+1];

	for(int i = 0; i < n; i++){
		tabla[i][0] = 0; // La primera columna
```

```
        }

        // Rellenamos tabla
        for(int i = 0; i < n; i++){
                for (int j = 1; j<=suma; j++){

                        int v1;                                         //
                        if ( i-1 < 0)                                   // Si no se
                                v1 = numeric_limits<int>::max();        //   coge
                        else                                            //
                                v1 = tabla[i-1][j];                     //

                        int v2;                                         //
                        if (j - usados[i] < 0)                          // Si se
                                v2 = numeric_limits<int>::max();        // coge
                        else                                            //
                                v2 = tabla[i][j-usados[i]] +1;          //

                        tabla[i][j] = min(v1,v2);
                }
        }


        /// Obtenemos la solucion

        int i = n-1;
        int j = suma;

        while( i >= 0 and j>0 ){

                int v1;
                if (i-1 <= 0)
                        v1 = numeric_limits<int>::max();
                else
                        v1 = tabla[i-1][j];

                int v2;
                if(j-usados[i] < 0)
                        v2 = numeric_limits<int>::max();
                else
                        v2 = tabla[i][j-usados[i]] +1;

                // Añadimos solucion a subsecuencia
                if (tabla[i][j] == v2){
                        subsecuencia.push_back(usados[i]);
                        j = j-usados[i];
                        i = i-1;
                }
                else
                        i = i-1;
        }
}
```

5. Eficiencia teórica:

La eficiencia para rellenar la tabla es de **Θ(n * M)** , es decir, lo que tardamos en recorrer toda la tabla (filas * columnas).

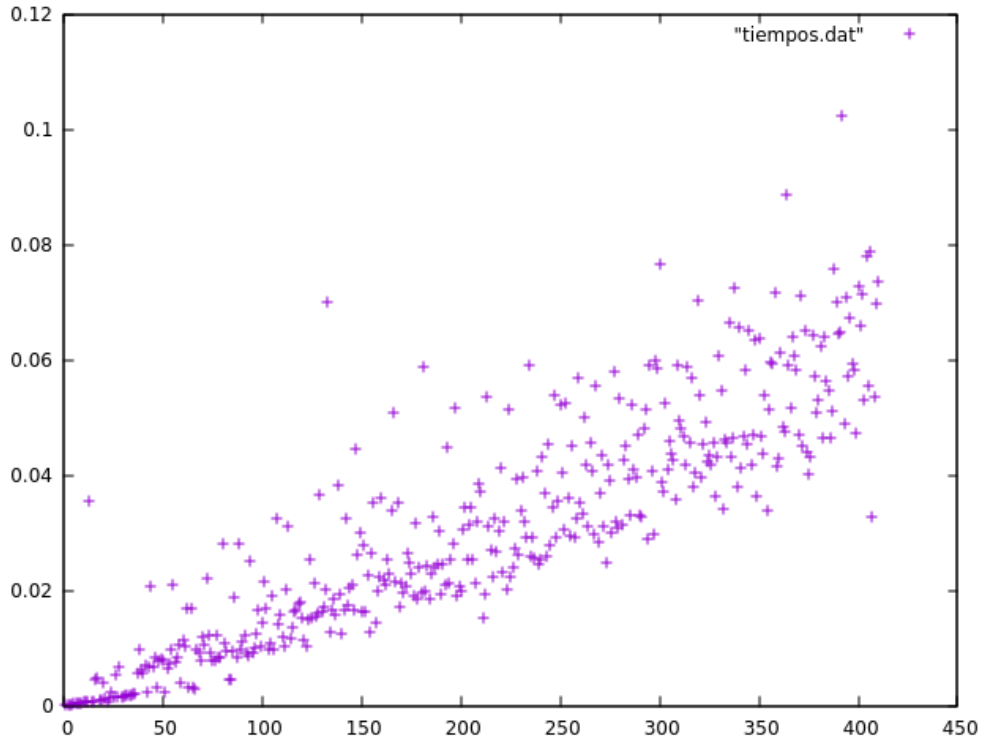
Para obtener la solución, en el peor de los casos tenemos una eficiencia de O(n + M).
Este caso es muy parecido al del problema planteado del cambio de moneda, la diferencia es que en este caso cuando encontramos solución directamente ascendemos a la fila de arriba, mientras que en el problema del cambio de monedas no.

Finalmente estos datos hacen que en el peor de los casos tengamos un orden de eficiencia de **O(n)**.

6. Eficiencia Empírica:

Para la eficiencia empírica he desarrollado un pequeño código para que funciona correctamente para distintos tamaños de vector (secuencia). Los valores tomados son.
M = 5000
La secuencia tiene desde 1 elemento hasta 410. Iremos incrementando de 1 en 1.
Estos datos los he tomado debido a que por las características del programa en sí, éste experimenta un desbordamiento de memoria si sobrepasamos estos valores, debido al gran tamaño de la tabla en memoria. Desconozco si puede ser por otro motivo pero todo apunta al desbordamiento por el motivo antes comentado, ya que no es algo aleatorio, sino que siempre ocurre si superamos cierto valor, lo que me hace comentar y decidir que estos son los valores con los que se puede realizar la eficiencia empírica.

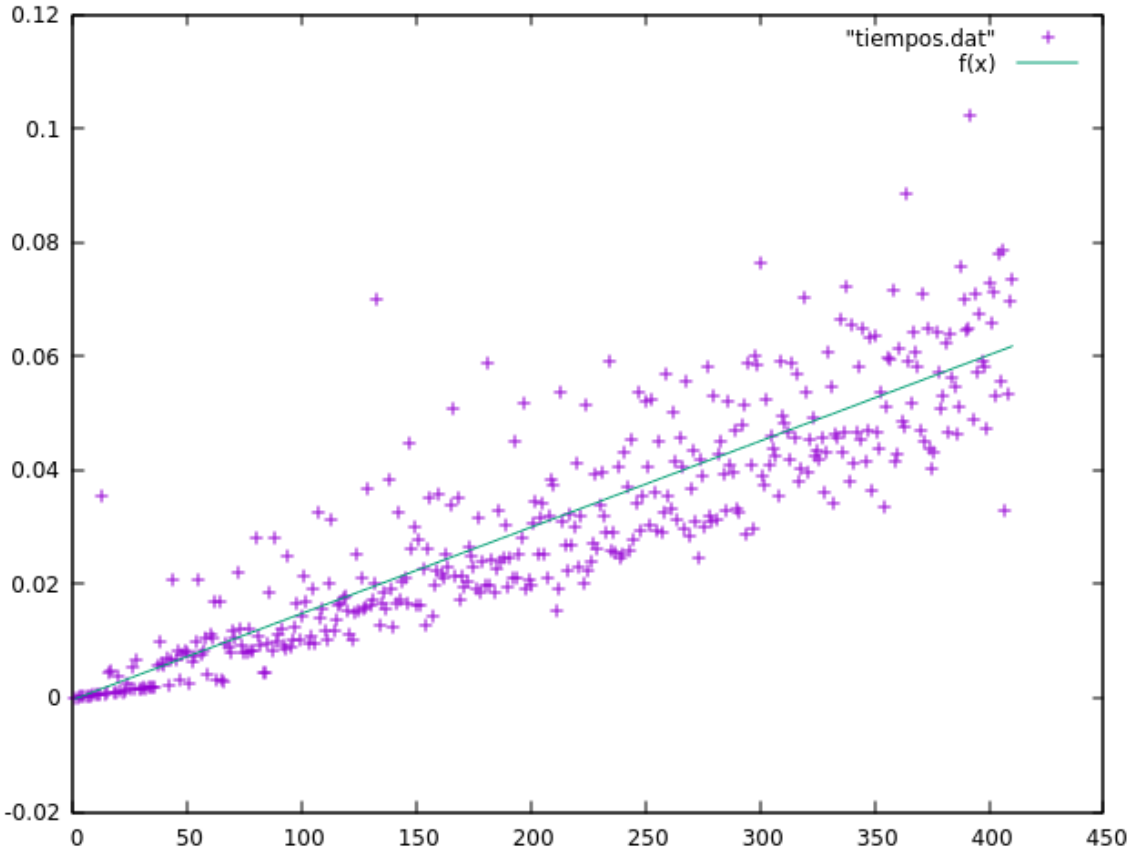
Los resultados son los siguientes:



El ajuste obtenido es el siguiente:

a = 0.000151327
b = -0.000300298

Tras ajustar obtenemos lo siguiente:



7. Dificultad:

La dificultad del ejercicio bajo mi punto de vista es de 6. Pese a ver que es un ejercicio sencillo en el que rápidamente vemos como debemos enfocarlo, el hecho de trabajar por primera vez con Programación Dinámica me resulta algo complejo.

Cifras
VORAZ

1. Datos del problema:

S: Conjunto de candidatos. Son el conjunto de números enteros que hay dentro del archivo .txt que pasemos como parámetro al programa. Representados en el código con la variable *secuencia*.

M: Es el valor que deben sumar. Representado en el programa por la variable *suma_final*.

Solución: Es una subsecuencia de S. Son los valores que sumados, restados, multiplicados o divididos dan como resultado M.

2. Representación de la solución:

Una Solución = (x_1OP, x_2OP, … , x_nOP) siendo $x_i = 0$ ó $x_i = 1$ donde x_i representa si se coge o no el elemento de la secuencia y donde OP representa la operación a realizar con dicho elemento.

- Si $x_i = 1$, se coge el elemento de la secuencia S.
- Si $x_i = 0$, no se coge el elemento.
- OP representa la correspondiente operación a realizar con el elemento escogido. A saber suma, resta, multiplicación o división.

3. Formulación matemática:

Minimizar $| M - \sum_{i=1}^{n} OPx_i \cdot S(i) |$
sujeto a la restricción
$x_i = 1$ ó $x_i = 0$

4. Diseño de la solución:

• Candidatos:

S es el conjunto de enteros con los que podemos operar.

• Función Solución:

Esta función en este ejercicio no está definida en sí misma. Esto es debido a que dentro de la función selección ya comprobamos si la subsecuencia encontrada es ya una solución.
Para saber si es solución, los elementos elegidos durante la selección, y que han sido añadidos al vector llamado *subsecuencia* junto con el operando correspondiente, deben de dar como resultado M.

- Función de Selección

En este caso no escogemos el valor más prometedor de la secuencia. El proceso aquí es el siguiente. Se escoge el primer elemento de la secuencia y se suma, se resta, se multiplica y se divide con el valor acumulado de los seleccionados previamente, que en el primer caso es 0 debido a que aún no se ha seleccionado ningún elemento de la secuencia S.
Una vez hecho esto comprobamos cual de los cuatro valores correspondientes a las cuatro operaciones está más cerca del valor que estamos buscando, es decir, M (supongamos que en este caso en particular M = 80).
Es fácil ver que la primera operación siempre va a ser la suma al tratarse de enteros positivos, por lo que supongamos que el primer elemento de la secuencia S es el 4 y el operando escogido es la suma. Bien, ahora en valor acumulado tenemos un 4, por lo que ahora escogemos el siguiente elemento de la secuencia S, supongamos que es el 3.
Volvemos a ejecutar la suma, la resta, la multiplicación y la división, y vemos que los resultados según los datos anteriores son 7 (suma), 1 (resta), 12 (multiplicación) y no tenemos en cuenta la división ya que el resto de la operación no es 0, por lo que vemos que el valor más cercano a 80 es el 12, lo que nos indica que el elemento 3 lo vamos a insertar en la solución junto al operador de multiplicación, y finalmente tendremos como valor acumulado 12.
Se ejecutará este proceso hasta que se encuentre la solución, o hasta que nos quedemos sin elementos en S.

- Función Factible

Esta función también está implícita dentro de la función de selección.
En este caso, hay veces que el elemento seleccionado no es factible. Esto ocurre por ejemplo en el siguiente caso. Supongamos que tenemos un valor acumulado de 78, y M = 80. y que el siguiente elemento es el 200. Podemos ver rápidamente que si operamos 78 con 200 el resultado va a quedar mucho más alejado de 80 de lo que 78 está, por tanto el operar en este caso nos perjudicaría, por lo que cuando el resultado de todas las operaciones está más alejado de la solución que el valor acumulado decidimos que el elemento en cuestión no es factible y se descarta de la solución. Por tanto, siempre que alguna de las operaciones nos acerque más al valor M buscado está función será true, mientras que si todas las operaciones nos alejan más de la solución será false.

- Añadir un objeto a la solución

Añadimos el valor si es factible.

- Función Objetivo

Suma, resta, multiplica o divide los valores de los candidatos escogidos den como resultado M.

La función de selección **no siempre encuentra el valor óptimo**. Si nos ajustamos y entendemos las funciones de selección y si es factible o no, podemos ver que se nos pueden acabar los elementos de S sin llegar a encontrar la solución. De hecho cuantos menos elementos tenga S más complicado será que encuentre la solución, debido a que cuanto más cerca estamos de la solución, más

concretas tiene que ser las operaciones y los elementos que necesitemos para llegar a la solución. Por ejemplo si tenemos un valor acumulado de 79 y M = 80 el algoritmo solo encontrará la solución final si en los elementos restantes de S hay un 1, para así sumarlo y llegar a la solución, ya que cualquier otro número nos alejaría de M y el algoritmo lo entendería como un valor no factible.

• Código

El código es el siguiente:

```
/**     @brief: Método que selecciona de un vector de enteros un subconjunto de ellos, cuyo
resultado según los operandos +,-,*,/ elegidos es igual al parámetro suma_final.
        @param secuencia: Conjunto de enteros.
        @param suma_final: valor entero que busca ser el resultado de las operaciones del
subconjunto elegido del conjunto secuencia.
        @param subsecuencia: es el subconjunto resultado, cuyos elementos con los operadores
seleccionados da como resultado el parámetro suma_final.
*/
void seleccion(const vector<int> &secuencia, int suma_final, vector<int> & subsecuencia){
        vector <int> usada = secuencia;
        cerr << "Llamado metodo seleccionar" << endl;

        int n = secuencia.size();       // Numero de elementos de la secuencia
        int cercano = numeric_limits<int>::max();   // Sera el numero mas cercano a la suma.

        int cercano_anterior = numeric_limits<int>::max();
        int acumulacion = 0;            // Es el valor entero que va aculumando la suma, resta,
                                        // multiplicacion o dividision de los valores enteros que
                                        // vayamos eligiendo
        int posicion;                   // Sera la posicion donde este el mayor valor, es decir, el valor
                                        // que pase a acumularse
        int suma;
        int resta;
        int multi;
        int division = numeric_limits<int>::max();
        bool su = false;
        bool re = false;
        bool mu = false;
        bool di = false;
        bool encontrado = false;

                for (int i = 0; i < n and !encontrado; i++){
                        suma = acumulacion + usada[i];          // Hacemos todas las
                                                                // operaciones por el valor
                        resta = acumulacion - usada[i];         // que vamos acumulando
                                                                // para despues comprobar
                        multi = acumulacion * usada[i];         // cual de ellos es el mas
                                                                // cercano al valor
                                                                // suma_final

                        if ( acumulacion % usada[i] == 0 and acumulacion >= usada[i])
                                division = acumulacion / usada[i];
```

```
        else                // Siempre que no se pueda le asignamos valor maximo
            division = numeric_limits<int>::max();      // para que no haya
                                                        // errores con
                                                        // interaciones
                                                        // anteriores

// Nos quedamos con la diferencia a suma_final, para asi saber luego cual es
// el mas cercano
        suma = suma_final - suma;
        resta = suma_final - resta;
        multi = suma_final - multi;
        division = suma_final - division;

// Pasamos a valores absolutos
        if (suma < 0)
            suma = -suma;
        if (resta < 0)
            resta = -resta;
        if (multi < 0)
            multi = -multi;
        if (division < 0)
            division = -division;

// Ahora vemos cual de todas ellas es la mas cercana a suma_final
        if (suma < cercano){
            cercano = suma;
            su = true;
            re = false;
            mu = false;
            di = false;
        }
        if (resta < cercano){
            cercano = resta;
            su = false;
            re = true;
            mu = false;
            di = false;
        }
        if (multi < cercano){
            cercano = multi;
            su = false;
            re = false;
            mu = true;
            di = false;
        }
        if (division < cercano){
            cercano = division;
            su = false;
            re = false;
            mu = false;
            di = true;
        }
```

```
                    // Una vez hemos hallado de nuestras operaciones el valor mas cercano
                    // pasamos a comprobar si nos aleja o nos acerca mas al valor buscado

                        if (cercano < cercano_anterior){

                            if(su){
                                cout << "+" << usada[i] << endl;
                                acumulacion = acumulacion + usada[i];
                            }
                            if(re){
                                cout << "-" << usada[i] << endl;
                                acumulacion = acumulacion - usada[i];
                            }
                            if(mu){
                                cout << "*" << usada[i] << endl;
                                acumulacion = acumulacion * usada[i];
                            }
                            if(di){
                                cout << "/" << usada[i] << endl;
                                acumulacion = acumulacion / usada[i];
                            }
                            cout << "Acumulacion: " << acumulacion << endl;
                            subsecuencia.push_back(usada[i]);
                            usada.erase(usada.begin()+i);
                            n--;    // Al borrar tenemos que indicar que el tamaño es
                                    // menor para que el bucle no se salga de memoria
                            i--;
                            cercano_anterior = cercano;
                        }
                        su = false;
                        re = false;
                        mu = false;
                        di = false;
                        cercano = numeric_limits<int>::max();

                        if (acumulacion == suma_final)
                            encontrado = true;
                    }
                    cout << endl;
}
```

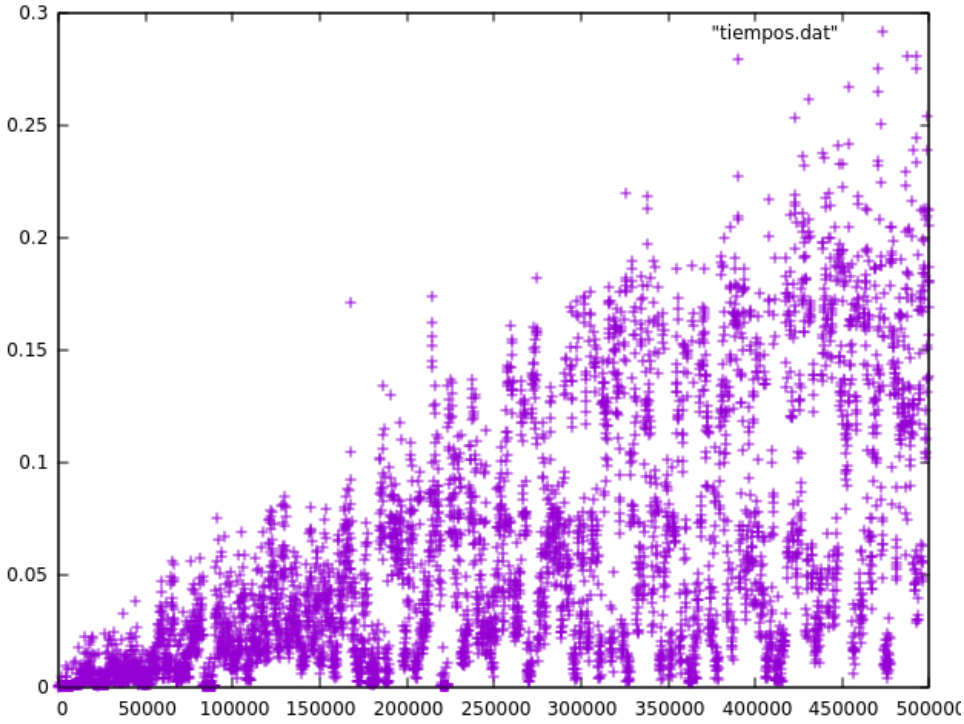
5. Eficiencia teórica:

Es fácil comprobar que en cualquiera de los casos el orden de eficiencia es O(n). Como hemos visto, el algoritmo va a recorrer el vector de elementos (secuencia S) desde el principio hasta el final. Por tanto, **su orden de eficiencia en el peor de los casos es O(n), al igual que en el mejor de los casos y en el caso promedio**.

6. Eficiencia Empírica:

Para calcular la eficiencia de forma empírica he realizado un código similar al usado en el primer ejercicio de algoritmos voraces, de modo que, ejecutamos varias veces al programa para tamaños de vector (secuencia) de 100 hasta 500000 con un incremento de 100 en cada iteración. El valor que buscamos vale 100000 en esta ocasión, debido a que si tratamos con numeric_limits<int>::max() podemos tener problemas debido a como trabaja el programa, el cual podría sobrepasar ese valor generando un error.

Los resultados obtenidos son los siguientes:

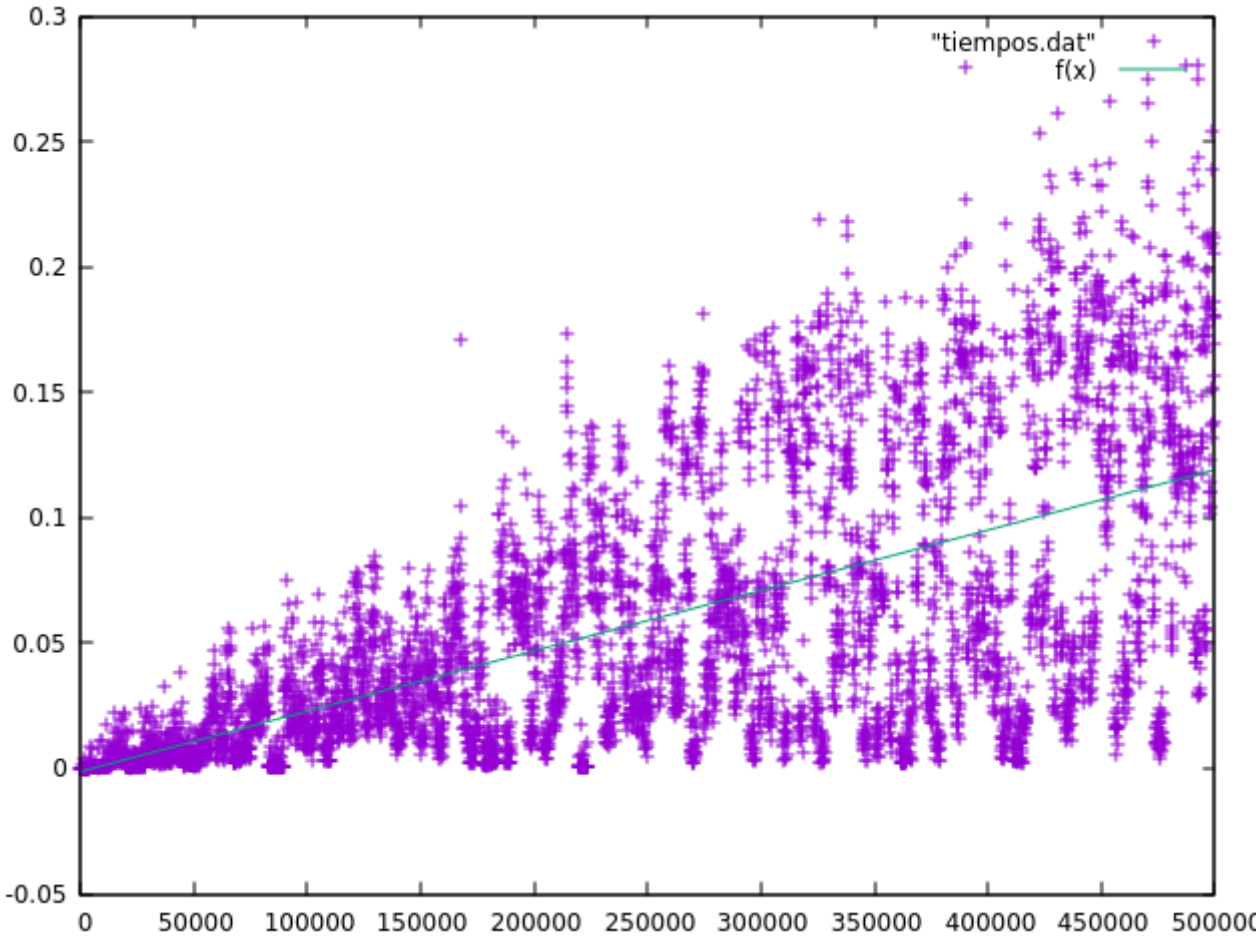


Vemos que los valores obtenidos, pese a que tiene forma lineal, son muy dispares. Esto es debido a como trabaja el algoritmo. Hay que comentar que los valores mas altos se dan cuando el algoritmo recorre todo el vector, mientras que los otros son debido a que encuentra solución antes de llegar a recorrer todo el vector.

El ajuste obtenido es:

a = 2.40775e-07
b = -0.00125673

Tras el ajuste obtenemos:



7. Dificultad:

La dificultad de este ejercicio ha sido de 6. La mayor complicación en este ejercicio considero que ha estado, por lo menos bajo mi punto de vista, en seleccionar y decantarse por un algoritmo voraz, es decir, como enfocar el problema mediante algoritmos voraces. Una vez tomada la decisión de como enfocar el ejercicio, la implementación es bastante sencilla.

Me gustaría añadir que tanto en este ejercicio como en el anterior (voraz), el hecho de realizar la formulación matemática me ha resultado un problema, ya que no sé como realizar una fórmula correcta que represente desde un punto de vista matemático mi problema, pero creo que las fórmulas que he aportado pueden explicar o entender lo que se busca con estos algoritmos.