Programa para Aproximar el Universo del Alfabeto Binario

Pérez Velázquez Leonardo Daniel[1]
Escuela Superior de Cómputo del Intituto Politécnico Nacional

19 de diciembre de 2023

[1]Boleta 2022630164

Índice general

Introducción

Uno de los objetivos que se tiene al estudiar teoría de la computación es conocer que son capaces de hacer las computadoras, y tambien que no pueden hacer. El programa que se diseñó y se analizó es un excelente caso para ese objetivo.

Antes de entrar de lleno a la explicación de cada paso que se siguió para llevar a cabo este programa, vale la pena dar un poco de teoría para entender la definición del algoritmo dentro del programa.

Es por esto que en el capítulo 1 hablaremos sobre lo que es un autómata. Dentro de su definición veremos las componentes que necesitamos entender para poder realizar el programa. Se debe destacar que cuando se mencione *"programa"*, se hace referencia al programa que se definirá brevemente al final del capítulo 1 y más formalmente en el capítulo 2.

Una vez entendido las componentes de un autómata, en el capítulo 2 podremos definir la entrada y salida del cerebro de nuestro programa, es decir, el algoritmo. Analizaremos su complejidad temporal y espacial, incluso el tamaño de la salida. Esto último será un gran sorpresa para quienes están acostumbrados a programar sin analizar y obvio para quienes tuvieron un un buen desempeño en materia de análisis y diseño de algoritmos.

Finalmente, en el capítulo 3 probaremos el programa y confirmaremos nuestro análisis teórico.

Capítulo 1

Autómata

Un *autómata* se puede definir como un dispositivo de cálculo abstracto. Esta definición "de diccionario" es corta y clara, pero no nos sirve para nuestro objetivo. Necesitamos una definición que permita poner al ente, autóma, sobre uno de los cimientos más fuertes construidos por el humano: las matemáticas.

1.1. Definición de un autómata

Se sabe que al menos existen dos tipos de autómatas, pero para entender el programa que se desea, basta con definir el autómata finito determinista tal como se hace en [2].

Un *autómata finito determinista* es un vector de 5 componentes

$$A = (Q, \Sigma, \delta, q_0, F)$$

donde

- Q es un conjunto finito de estados.
- Σ es un conjunto finito de símbolos de entrada.
- δ es una función de transición que toma como argumentos un estado y un símbolo de entrada y devuelve un estado.
- q_0 es un estado inicial tal que $q_0 \in Q$.
- F es un conjunto de estados finales tal que $F \subseteq Q$

A la notación anterior se le llama "quíntupla".

El componente que usaremos para el programa será Σ, el conjunto de símbolos de entrada. Por lo que solo nos enfocaremos en este.

1.1.1. Alfabeto

A un conjunto de símbolos finito y no vacío se le conoce como *alfabeto*. Por convención se utiliza el símbolo sigma para designar un alfabeto.

Definamos el alfabeto *binario* como

$$\Sigma = \{0, 1\}$$

Este alfabeto es el que se necesita para entender el problema al que se enfrentó él programa.

1.1.2. Cadenas de caracteres

Dado un alfabeto, podemos construir entes conocidos como cadenas de caracteres.

Una *cadena de caracteres* o *palabra* es una secuencia finita de símbolos seleccionados de algún alfabeto.

La lista siguiente contiene algunas cadenas del alfabeto binario $\Sigma = \{0, 1\}$.

- 00000.
- 0101.
- 1.

La cadena vacía

Hay una cadena que, aunque sea trivial, no es muy intuitiva. La cadena vacía.

La *cadena vacía* es aquella cadena que presenta cero apaciriciones de símbolos. Se designar por ε. Tiene la propiedad unica que puede construirse en cualquier alfabeto.

Longitud de una cadena

Para poder hacer un análisis del algoritmo que se diseñó necesitamos definir la longitud de una cadena.

La *longitud* de una cadena w es el número de posiciones ocupadas por símbolos dentro de w y se denota como $|w|$. De este modo, $|01| = 2$.

Una propiedad interesante que tiene la cadena vacía es la siguiente.

$$|\varepsilon| = 0$$

1.1.3. Potencias de un alfabeto

Se puede expresar el conjunto de todas las cadenas de una determinada longitud de un Σ utilizando una notación exponencial.

Se define Σ^k como el conjunto de las cadenas de longitud k, tales que cada uno de los símbolos de las mismas pertenece a Σ.

De esta manera, si $\Sigma = \{0, 1\}$ tenemos que $\Sigma^0 = \{\varepsilon\}$, $\Sigma^1 = \{1, 0\}$, $\Sigma^2 = \{00, 01, 10, 11\}$, $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$, etc.

Conjunto de todas las cadenas de un alfabeto

Un conjunto interesante es aquél que contiene cadenas de un alfabeto pero no solo de una determinada longitud, si no de varias longitudes.

El conjunto de todas las cadenas de un alfabeto sigma se designa mediante Σ^*. Mas formalmente

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots = \bigcup_{i=0}^{\infty} \Sigma^i$$

Por consiguiente

$$\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \dots\}$$

1.2. Conjunto Σ_n^*

Ahora que se ha comprendido los conceptos sobre autómatas que se necesitan para poder diseñar y crear el programa, se puede definir lo que será el objetivo de este.

Por convenio, en este reporte definimos Σ_n^* como sigue.

$$\Sigma_n^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \cup \Sigma^{n-1} \cup \Sigma^n = \bigcup_{i=0}^{n} \Sigma^i$$

Observese que

$$\Sigma_n^* \subset \Sigma^*$$

Por ejemplo, $\{0, 1\}_2^* = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$ y $\{0, 1\}_0^* = \{\varepsilon\}$

¿Qué realizará el programa?

"Simplemente", dado un número entero n, computar Σ_n^*.

Capítulo 2

El programa

Lo que hace poderoso un programa de computadora es el algoritmo que "lleva dentro". Sabemos que una de las propiedades de los algoritmos es que deben ser *generales*, es decir, deben funcionar para todas las entradas.

Es común ver algoritmos cuyas entradas son complejas, una sola entrada puede consistir en un vector de millones de componentes desordenados. Tambien se dice que los pasos finitos y concretos que siguen los algoritmos son tan abstractos y generales que se necesita un nivel de organización y de entendimiento profundo por parte de la persona que diseño y analizó el algoritmo.

Ante esto, muchas veces las salidas de los algoritmos se toman como algo que no afecta el objetivo de un algoritmo. Este programa probará que un algoritmo puede tener una entrada sencilla, y pasos muy intuitivos que los podría entender cualquier persona, pero una salida tremendamente engorrosa. Una salida que haría que los programadores confiados en datos precomputados o en la programación dinamica prefieran utilizar el procesador.

2.1. Programa para procesar Σ_n^*

El programa que se diseñó y analizó, procesa Σ_n^*, es decir, muestra un determinado número de cadenas de un alfabeto Σ, específicamente del alfabeto

$$\Sigma = \{0, 1\}$$

A continuación se definen las funcionalidades que debe tener a partir de su capacidad para procesar Σ_n^*.

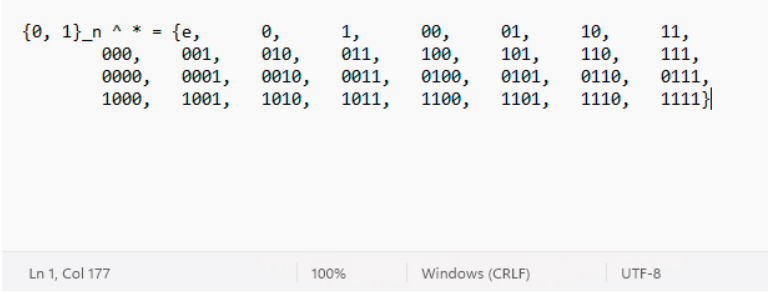


Figura 2.1: Contenido esperado para el archivo de salida.

- Entrada de da datos
- Salida de datos
- Crear gráficas

Ahora se detalla cada una de estas funcionalidades.

2.1.1. Entrada

El programa debe preguntar por un n_1, para después procesar Σ_n^* con $n = n_1$. A continuación pregunta por otra n_2, y procesa Σ_n^* ahora con $n = n_2$, y así sucesivamente hasta que se le especifique.

La entrada n del programa es tal que $n \in [0, 1000]$ y $n \in \mathbb{Z}^+ \cup \{0\}$

2.1.2. Salida

La salida debe ser expresada en notación de conjunto y se debe guardar en un archivo de texto.

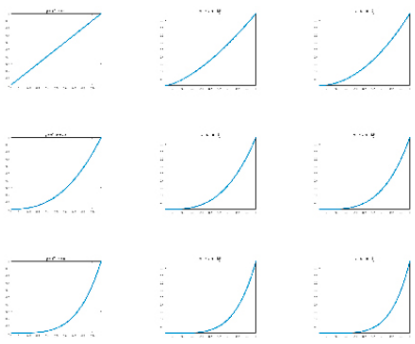
En la figura 2.1 se observa el contenido que se espera que tenga el archivo de salida al "pedirle" al programa que procese Σ_n^* para $n = 4$.

Notese que el puntero de posición en el archivo está en la linea 1, columna 177. Esto significa que no debe haber saltos de linea, si no que toda la definición del conjunto debe estar en una sola linea.

Como se puede apreciar en la figura 2.1, se usa e para representar la letra griega ε, la cadena vacia.

2.1.3. Gráficar

Del archivo de salida, graficar el número de unos de cada cadena. Cada valor en el eje de las x representa la posición de las cadenas en el conjunto



Save the nine images into a GIF file. Because three-dimensional data is not supported for GIF files, call `rgb2ind` to convert the RGB data in the image to an indexed image `A` with a colormap `map`. To append multiple images to the first image, call `imwrite` with the name-value argument `WriteMode` set to `"append"`.

```
filename = "testAnimated.gif"; % Specify the output file name
for idx = 1:nImages
    [A,map] = rgb2ind(im{idx},256);
    if idx == 1
        imwrite(A,map,filename,"gif","LoopCount",Inf,"DelayTime",1);
    else
        imwrite(A,map,filename,"gif","WriteMode","append","DelayTime",1);
    end
end
```

Figura 2.2: Documentación para gráficar en el software MATLAB.

y el eje de las y el número de unos que tienen esas cadenas.

De igual modo, se generará otra gráfica, pero considerando al eje y como el logaritmo base 10 de el número de unos que tienen las cadenas.

Estas gráficas tambien forman parte de la salida de el programa, sin embargo, el archivo con Σ_n^* es la principal salida de el programa por lo que se detallará más el algorimo para crear esta salida, pues la generación de la gráfica la realizará otro programa especializado, MATLAB (vease figura 2.2).

2.2. Algoritmo para computar Σ^n

Una estrategia popular y que se cree muy efectiva para atacar y resolver un problema consiste en separarlo en subproblemas, modularizarlo. Es una estrategia por excelencia para diseñar, analizar y entender un algoritmo.

El problema de procesar Σ_n^* puede modularizarse gracias a como está definido. Recordemos su definición:

$$\Sigma_n^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \cup \Sigma^{n-1} \cup \Sigma^n = \bigcup_{i=0}^{n} \Sigma^i$$

Esta definición ya nos esta "diciendo" como modularizar el problema: computar Σ^n para $n = 1, 2, 3, \ldots, n-1, n$ y unir estos conjuntos.

2.2.1. El problema de numeros binarios

Como indicamos en el capitulo 1, el alfabeto para el cual calculamos Σ_n^* es el alfabeto binario

$$\Sigma = \{0, 1\}$$

Si calculamos manualmente $\{0, 1\}^n$ para $n = 2$, tenemos que

$$\{0, 1\}^2 = \{00, 01, 10, 11\}$$

Si se considera a cada cadena de $\{0, 1\}^2$ como un número entero base 2, entonces un conjunto equivalente a $\{0, 1\}^2$ se define a continuación.

$$\{0, 1\}^2 = \{00, 01, 10, 11\} \equiv \{00_2, 01_2, 10_2, 11_2\} = \{0_{10}, 1_{10}, 2_{10}, 3_{10}\}$$

Se observa que $\{0, 1\}^2$ es equivalente al conjunto de los primeros 2^2 números del conjunto $\mathbb{Z}^+ \cup \{0\}$.

Del mismo modo, para $n = 3$, tenemos que

$$\begin{aligned} \{0, 1\}^3 &= \{000, 001, 010, 011, 100, 101, 110, 111\} \\ &\equiv \{000_2, 001_2, 010_2, 011_2, 100_2, 101_2, 110_2, 111_2\} \\ &= \{0_{10}, 1_{10}, 2_{10}, 3_{10}, 4_{10}, 5_{10}, 6_{10}, 7_{10}\} \end{aligned}$$

Donde $\{0, 1\}^3$ es equivalente al conjunto de los primeros 2^3 números del conjunto $\mathbb{Z}^+ \cup \{0\}$.

Generalización

En general, $\{0, 1\}^n$ es equivalente al conjunto de los primeros 2^n números del conjunto $\mathbb{Z}^+ \cup \{0\}$.

Calcular números y no cadenas

Se ha modularizado y abstraido el problema de modo que en vez de calcular cadenas, calculemos números, esto es de gran ayuda pues algorimos como el de *obtener el sucesor de un número* son conocidos y sencillos de implementar en cualquier lenguaje de programación.

2.2.2. Algoritmo para números enteros base 2

El algoritmo que sera el cerebro del programa será aquel que pueda computar el conjunto de los primeros 2^n números del conjunto $\mathbb{Z}^+ \cup \{0\}$.

A continuación se define el algoritmos para números enteros base 2.

Entrada: Un número entero $0 \leq n \leq 1000$.

Salida: Un vector ordenado E cuyos elementos son los primeros 2^n números base 2 del conjunto $\mathbb{Z}^+ \cup \{0\}$, tal que $|E| = 2^n$.

Elección del algoritmo

Hay por lo menos dos maneras de crear el algoritmo.

Sucesores base 10 y equivalencia en base 2. Esta manera es muy intuitiva, consiste en empezar con el número 0 y calcular sus $2^n - 1$ sucesores. Todo esto utilizando números base 10. Para cada sucesor de 0 se deberá guardar su equivalencia númerica en base 2. Las 2^n equivalencias númericas de los sucesores del 0, incluyendolo, serán la salida del algorimo. De este modo solo necesitamos el algoritmo para obtener la equivalencia en base 2 de un número base 10.

Sucesores base 2. Esta manera un poco más "complicada" que la anterior. Consite en obtener los $2^n - 1$ sucesores de 0 en base 2. Es decir, dado algun sucesor del número 0 base 2, se debe sumar 1 a ese número, realizando esta operación en base 2 para obtener un número en igual base. A diferencia del algorimo anterior, se neceista un algorimo para sumar 1 a un número base 2.

Los dos caminos. Se tiene, por un lado, diseñar un algoritmo para convertir un número base 10 a base 2. Por otro lado, diseñar un algoritmo para sumar 1 a un número base 2. La elección es bastante sencilla de hacer porque se sabe que para convertir un número base 10 a base 2 se necesitan operaciones "complejas" para el procesador, como lo es la operación modulo y la operación división. Para sumar 1 a un número base 2 no se necesita ni siquiera la operación suma (impresionantemente).

Diseño del algoritmo

Aquí es donde se confirma el poder de modularizar un problema, pues el problema original se ha dividido y abstraído hasta un punto donde solo tenemos que diseñar un algoritmo que sume 1 a un número base 2.

Aunque sería lo mejor, no se diseñó el algoritmo pensando en tomar el valor de una posición del número base 2 como un bit de manera física en la computadora. Esto porque la implementación sería muy engorrosa. Se diseñó el algoritmo a un nivel más alto de abstracción en cuanto a implementación se refiere.

De este modo se considera un número base 2, β_2, como un vector $N = (a_0, a_1, a_2, a_3, \ldots, a_{n-1}, a_n)$ con $a_0, a_1, a_2, a_3, \ldots, a_{n-1}, a_n \in \{0, 1\}$ tal que

$$\beta_2 = a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_3 2^3 + a_2 2^2 + a_1 2^1 + a_0 2^0$$

Ahora definamos la operación de sumar 1 a un número base 2 como sigue.

$$\beta_2 + 1 = s_n 2^n + s_{n-1} 2^{n-1} + \cdots + s_3 2^3 + s_2 2^2 + s_1 2^1 + s_0 2^0$$

Donde

$$\begin{aligned} s_0 &= a_0 \veebar 1 \\ &= \neg(a_0 \leftrightarrow 1) \\ &= \neg((a_0 \rightarrow 1) \wedge (1 \rightarrow a_0)) \\ &= \neg((\neg a_0 \vee 1) \wedge (0 \vee a_0)) \\ &= \neg(1 \wedge a_0) \\ &= \neg a_0 \end{aligned}$$

Además

$$\begin{aligned} s_1 &= a_1 \veebar (a_0 \wedge 1) \\ &= a_1 \veebar a_0 \\ s_2 &= a_2 \veebar (a_1 \wedge (a_0 \wedge 1)) \\ &= a_2 \veebar (a_1 \wedge a_2) \\ s_3 &= a_3 \veebar (a_2 \wedge (a_1 \wedge (a_0 \wedge 1))) \\ &= a_3 \veebar (a_2 \wedge (a_1 \wedge a_0)) \\ &= a_3 \veebar (a_2 \wedge a_1 \wedge a_0) \\ &\vdots \\ s_n &= a_n \veebar (a_{n-1} \wedge a_{n-2} \wedge \cdots \wedge a_3 \wedge a_2 \wedge a_1 \wedge a_0) \end{aligned}$$

Escrito de otra forma, se tiene que

$$s_n = a_n \veebar \bigwedge_{i=0}^{n-1} a_i$$

En [3] se demuestran todas las equivalencias lógicas usadas en las igualdades anteriores y a lo largo del reporte. Notese que no se considera s_{n+1}, aunque exista. Esto se debe a que todas las cadenas en Σ^n tienen la misma longitud (n).

Como la naturaleza de s_n es cíclica, diseñar el pseudocodigo de el algoritmo será directo, pues bastara con tener un ciclo bastante usual. En el siguiente *pseudocodigo* queda definido el algorimto para sumar 1 a un número base 2.

```
1  SUCESOR(N):
2       i = 0
3       conjuntion = 1
4
5       prev = N[0]
6       N[0] = N[0] EXOR conjuntion
7       i = i + 1
8       while conjuntion != 0 AND i < |N|:
9            conjuntion = conjuntion AND prev
10           prev = N[i]
11           N[i] = N[i] EXOR conjuntion
12           i = i + 1
```

Notese que una de las condiciones para deter el ciclo de la linea 8 es que $\bigwedge_{i=0}^{k-1} a_i = 0$ para algun $k \leq n$. El por qué se decidio agregar esa condicion es para limitar el numeros de veces que se entrá al ciclo y por tanto hacer las eficiente. Esto no alterá la lógica anterior pues si $\bigwedge_{i=0}^{k-1} a_i = 0$, entonces

$$(\bigwedge_{i=0}^{k-1} a_i) \wedge p = 0 \wedge p = 0 \Longrightarrow \bigwedge_{i=0}^{n-1} a_i = 0$$

también

$$\begin{aligned} a_n \veebar \bigwedge_{i=0}^{n-1} a_i &= a_n \veebar 0 = \neg(a_n \leftrightarrow 0) \\ &= \neg((a_n \rightarrow 0) \wedge (0 \rightarrow a_n)) = \neg((\neg a_n \vee 0) \wedge (1 \vee a_n)) \\ &= \neg(\neg a_n \wedge 1) = \neg(\neg a_n) \\ &= a_n \end{aligned}$$

Con este algoritmo es facil diseñar el algoritmo prinicipal de el programa, pues basta con computar cada sucesor de 0 en base 2 hasta 2^n y guardarlos. El pseudocodigo de este algoritmo se muestra a continuación.

```
1  STRINGS(n):
2       i = 0
3       while i < n:
4            N[i] = 0
5
6       SAVEN(N)
7       j = 1
8       while j <= 2 ^ n - 1:
9            SUCESOR(N)
10           SAVEN(N)
```

La funcion $SAVEN()$ guarda el vector N en un archivo en el disco duro.

Implementación del algoritmo

Ya que tenemos el pseudocodigo de el algortimo principal del programa, es muy sencillo implementarlo. Se decidió implementarlo con el lenguaje de programación C, porque como se dijo anteriormente no necesitamos un nivel de abstraccion muy bajo, pero tampoco alto.

En las listings 2.1 y 2.2 se muestran solamente la implementación de las dos funciones definidas anteriormente. El codigo completo del programa se muestra en el apendice A. Se decidio no analizar los algoritmos para todas las funcionalidades porque se consideran triviales y muchas de ellas se profundizan en [1].

```
10 void Sucesor(int* N, int N_longitud)
11 {
12     int conjuncion = 1;
13
14     int previo = *(N);
15     *(N) = EXOR(*(N), conjuncion);
16     int i = 1;
17     while(conjuncion != 0 && i < N_longitud)
18     {
19         conjuncion = AND(conjuncion, previo);
20         previo = *(N + i);
21         *(N + i) = EXOR(*(N + i), conjuncion);
22         i++;
23     }
```

```
24 }
```

Listing 2.1: Codigo en alfabeto.c del algorimo para obtener el sucesor de β_2.

```
276 int Computar(int n)
277 {
278     archivo_principal = fopen(nombre_archivo_principal, "wt");
279     if(archivo_principal == NULL)
280         return 1;
281 
282     fprintf(archivo_principal, "{0,␣1}_n␣^␣*␣=␣{e");
283 
284     for(int i = 1, dos_pot_i = 2; i <= n; i++, dos_pot_i *= 2)
285         Strings(i, dos_pot_i);
286 
287     putc('}', archivo_principal);
288 
289     fclose(archivo_principal);
290     return 0;
291 }
292 
293 void Strings(int n, int dospotn)
294 {
295     int* N = (int*)malloc(sizeof(int) * n);
296     if(N == NULL)
297         return;
298     for(int i = 0; i < n; i++)
299         *(N + i) = 0;
300 
301     Guardar(N, n);
302     for(int i = 1; i <= dospotn - 1; i++)
303     {
304         Sucesor(N, n);
305         Guardar(N, n);
306     }
307     free(N);
308 }
```

Listing 2.2: Parte del codigo en universo.c de la implementacion del algoritmo para obtener Σ_n^*.

Operaciones ideales. En este reporte se dice que una operacion que el programa usa para su funcionamiento y no "explicitamente" por el algoritmo principal es una operacion "desprecialble", esta solo funciona como ayuda a los 2 algorimos definidos en el diseño. Se asumira que cuando el programa realize una operacion como esta, nunca abra ningún error. Esto con el fin de

hacer más claro el código de la implementación. Solo en ciertos casos valdrá la pena tener un *control de errores*.

Análisis del algoritmo

Dada una implemetanción de algun programa, es muy importante el análisis del algoritmo que se usa. El análisis permite conocer cuánto tiempo tardara en procesar cierta petición, cuanta RAM se necesita, y en este caso, cuanta memoria externa ocupará la salida.

Análisis temporal

Empecemos analizando la función

```
int Computar(int n)
```

Como se observa en el listing 2.2, todas las operaciones son de complejidad constante, excepto el ciclo de la linea 200. El total de operaciones significativas (comparación que involucre a n) está dada por

$$n + 1 \tag{2.1}$$

Ahora, veamos la función tambien en unvierso.c

```
void Strings(int n, int dospotn)
```

Se observa que la mayor parte de operaciones tambien son de complejidad constante, sin embargo el ciclo de la linea 218 no. El número de operaciones significativas estara dada por la siguiente formula

$$\sum_{i=1}^{n+1} 2^i \tag{2.2}$$

Este es el ciclo que mayor "peso" tiene en el algoritmo. Finalmente veamos la funcion en alfabeto.c

```
void Sucesor(int* N, int N_longitud)
```

Todas las operaciones de esta funcione tienen complejidad constante menos el ciclo de la linea 14. En el peor de los casos realizará el siguiente numero de operaciones significativas

$$\sum_{i=1}^{n+1}((2^i - 1)i) \tag{2.3}$$

Obteniendo la funcion complejidad $f(n)$ con las ecuaciones (2.1), (2.2) y (2.3), tenemos que

$$f_c(n) = n + 1 + \sum_{i=1}^{n+1} 2^i + \sum_{i=1}^{n+1}((2^i - 1)i)$$

Desarrollando

$$\begin{aligned} f_c(n) = f(n) &= n + 1 + \sum_{i=1}^{n+1} 2^i + \sum_{i=1}^{n+1}((2^i - 1)i) \\ &= n + 1 + 2^{n+2} - 2 + \frac{1}{2}(-n^2 + (2^{n+3} - 3)n + 2) \end{aligned}$$

Simplificando, se tiene

$$f_c(n) = \frac{1}{2}(2^{n+3} - n)(n + 1)$$

Por tanto, la cota O del algoritmo completo es la siguiente.

$$O(2^n n)$$

Con esto se tiene que el tiempo que le lleva al algoritmo computar Σ_n^* crece exponencialmente conforme crece n.

Tiempo. Con en analisis anterior podemos aproximar el tiempo de computo del programa dada una entrada. Si se toma el valor máximo definido, $n = 1000$, como entrada, entonces el algoritmo realizará

$$\begin{aligned} &429032046317381435307749389643624724948786486 60 \cdot \cdot \cdot \\ &689565642047765550348856087042442344627663087 78 \cdot \cdot \cdot \\ &046215942889070361882799888049329723912025430 56 \cdot \cdot \cdot \\ &822486090930935149540093703962208656259827186 69 \cdot \cdot \cdot \\ &734052079328545340798085834262092834664665306 89 \cdot \cdot \cdot \\ &883796009255990994307069665691483931236104593 71 \cdot \cdot \cdot \\ &08044896171494949281004 \; op \end{aligned}$$

Considerando que un programa diseñado con el lenguaje de programación C ejecutado en una computadora promedio realiza $\frac{10^9}{2}\frac{op}{s}$, se tiene que, el tiempo para procesar $\{0,1\}^*_{1000}$ es aproximadamente ¡10^{288} años!

Análisis espacial

En las 3 funciones en los listings 2.1 y 2.2 se observa que en donde se crea memoria dinámica considerable es en el cuerpo de la función

```
void Strings(int n, int dospotn)
```

Específicamente en la linea 211 la cantidad de bytes que se estan reservando son

```
sizeof(int) * n
```

Donde normalmente un tipo de dato int utiliza 4 bytes. Por tanto, en el peor caso (cuando $n = 1000$) se usará

$$4 \cdot 1000 \; byte = 4000 \; byte$$

Lo que confirma que en este algoritmo se prefierio costo de procesamiento y no de memoria.

Vale la pena decir que existen más algoritmos que resuelven el problema que prefieren costo de memoria y no de procesamiento.

Tamaño de la salida

En el análisis espacial se obtuvo que el programa ocupará un máximo de 4000 bytes para su ejecución, sin embargo ¿pasa lo mismo con el tamaño de la salida?

Para este análisis se considerará que cada dígito que forma parte de la definición escrita de Σ_n^* ocupa $1byte$ en el disco duro.

Determinemos primero los bytes que se necesitan para representar Σ^n , pero sin los símbolos ‘’ y ‘’ y considerando que después de cada número se necesita la cadena de dos símbolos “,␣”. Por lo dicho en la definición del problema de números binarios, tenemos que la representación de Σ^n ocupa

$$2^n \cdot n + 2 \cdot 2^n \; byte$$

$2^n \cdot n$ por los digitos de todos los elementos de Σ^n y $2 \cdot 2^n$ por los “,␣” de cada elemento.

Ahora, como

$$\Sigma_n^* = \bigcup_{i=0}^{n} \Sigma^i$$

La función $M(n)$, que nos dice la cantidad de bytes que ocuparan todos los elementos de Σ_n^*, las cadenas “,␣”, los 2 simbolos ‘{’ y ‘}’ y la cadena

“{0,␣1}␣n␣^␣*␣=␣” esta definida como sigue

$$M(n) = \left(15 + 1 + 3 + \sum_{i=1}^{n}(2^i \cdot i + 2 \cdot 2^i) - 2 + 1\right) \ byte$$

Desarrollando

$$M(n) = (15 + 1 + 3 + 2(2^n n + 2^n - 1) - 2 + 1) \ byte$$

Simplificando

$$M(n) = (2^{n+1}(n + 1) + 16) \ byte$$

Notese que en la expresion antes de desarrollarla se tiene un -2, esto por que el ultimo elemento no debe tener la cadena “,␣”, simplemente tiene el caracter ‘{’.

Predecir. Tener $M(n)$ es de muy gran ayuda pues permite “predecir” cuanta memoria ocuapará la salida. Si evaluamos $M(n)$ junto con $f_c(n)$ dado un n, entonces se puede tomar una decisón sobre si proceguir con el computo o no.

Tomando $n = 1000$, tenemos que

$$M(1000) = (2^{1000+1}(1000 + 1) + 16) \ byte$$

Desarrollando, obtendremos que la salida coupará

$$21451602315869071765387469482181236247439324330^{3}\cdot\cdot\cdot$$

$$44782821023882775174428043521221172313831543890^{2}\cdot\cdot\cdot$$

$$31079714445351809413999440246648619560127152841^{1}\cdot\cdot\cdot$$

$$24304546546757477004685198110432812991359334867^{0}\cdot\cdot\cdot$$

$$26039664272670399042917131046417332332653449418^{9}\cdot\cdot\cdot$$

$$80046279954971535348328457419656180522968554022^{4}\cdot\cdot\cdot$$

$$48085747474890768 \ byte$$

Esto es más que 10^{274} Qbyte, es decir quintillones de bytes, el prefijo más grande para potencias de 10. Esta cantidad es excesiva y no hay memoria en el mundo que pueda almacenar tal salida.

Capítulo 3

Ejecución del programa

Un programa ejecutado desde consola normalmente funciona con comandos que son su entrada, los procesa y ejecuta una función. En este programa se siguió este enfoque para que el programa realice las funcionalidades descritas anteriormente.

3.1. Poniendolo a prueba

El programa tiene la capacidad para reconocer 4 comandos:

- ayuda
- n=x
- graficar
- salir

Donde x es un número entero entre 0 y 1 inclusive, o la cadena "aleatorio".

A continuación se hará probar cada comando.

Comando ayuda

Este comando es bastante básico. Al dar como entrada la cadena "ayuda" al programa este imprime todos los comandos junto con su descripción (vease figura 3.1). Tambien indica que entrada se considra como el valor lógico '1'.

Comando n=x

Este es el comando que hace que el programa ejecute la función principal. x puede ser un número entero entre 0 y 1 inclusive, o la cadena "aleatorio". Si x es un número, entonces el programa computará Σ_n^* con $n = x$. Si x es la cadena "aleatorio", entonces el programa generará un número pseaudoaleatorio p y despues computara Σ_n^* con $x = p$.

En la figura 3.2 se ovserva que cuando se introduce un número, el programa solicita una confirmación. Para confirmar se debe introducir '1' que es el valor logico 1. El El programa es capaz de calcular la memoria que ocupará el archivo de salida del algoritmo principal. La implementación del algoritmo que calcula la memoria en bytes que ocupara el archivo final se muestra en el listing 3.1, observe que está evaluando $M(n) = (2^{n+1}(n+1) + 16)$ *byte*, esto se obtuvo en el análisis del algoritmo que usa el programa.

```
32 char* M(int n)
33 {
34     ent resultado;
35     IniciarEnt(&resultado, "1");
36 
37     ent aux;
38     IniciarEnt(&aux, "0");
39 
40     for(int i = 0; i < n + 1; i++)
41     {
42         MultiplicaEntInt(resultado, 2, &aux);
43         CompiaEnt(aux, &resultado);
44     }
45 
46     MultiplicaEntInt(resultado, n + 1, &aux);
47     CompiaEnt(aux, &resultado);
48 
49     ent sum16;
50     IniciarEnt(&sum16, "16");
51     SumaEnt(resultado, sum16, &aux);
52     CompiaEnt(aux, &resultado);
53 
54     char* resultado_cadena = EntACad(resultado);
55 
56     LiberarEnt(&resultado);
57     LiberarEnt(&aux);
58     LiberarEnt(&sum16);
59     return resultado_cadena;
60 }
```

Listing 3.1: Codigo en alfabeto.c del algorimo para obtener la memoria que ocupará el archivo de salida.

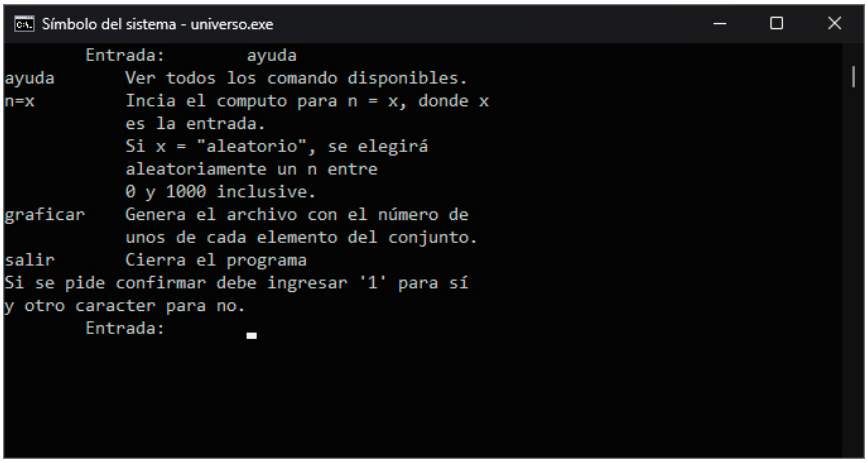


Figura 3.1: Probando el programá con el comando para ayuda.

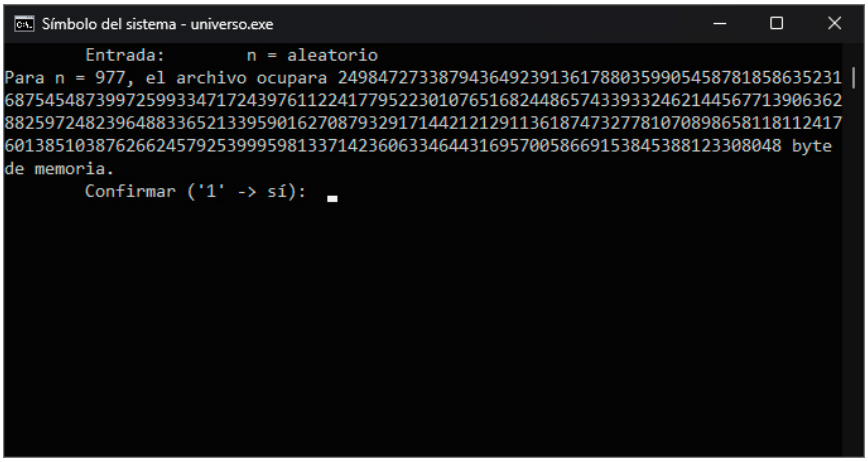


Figura 3.2: Probando el programá con el comando n=aleatorio.

La implementación de este algoritmo necesita de un tipo de dato para números enteros que soporte cantidades extremadamente grandes. Por esto es que se programó en la librería mientero.c este tipo de dato llamado

```
ent
```

Este tipo de dato puede almacenar números enteros de hasta 19327352823 dígitos.

Como se observa en la figura 3.2, para n = 977 el tamaño en bytes que ocupará el archivo de salida será descomunal. Por lo que para probar el programa, haremos que computé Σ_n^* para $n = 28$. En la figura 3.3 se observa que después de confirmar la entrada, el programa imprime el tiempo en segundos desde el 1 de enero de 1970, después nos confirma que está computando sigma y cuando termina de computar lo indica imprimiendo la cadena "Computo terminado" y de nuevo el tiempo en segundos desde 1970. Lo que más importa en esta salida en consola es Δt, que es al diferencia entre los dos tiempos que imprimió antes. Esto nos confirma que el programa tardo $\Delta t = 1377\ s$ o 23 minutos y 57 segundos en computar Σ_n^*. Para comprobar que la salida es correcta se observan las propiedades del archivo y se confirma que pesa exactamente lo que el programa nos indicó (vease figura 3.4).

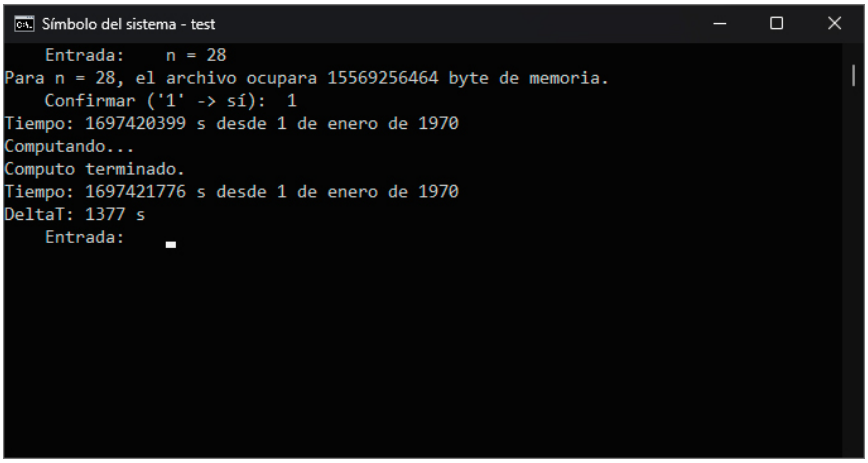


Figura 3.3: Probando el programá con el comando n=28.

Comando graficar

El comando graficar genera un archivo tipo CSV, donde cada elemento representa el número de 1s que contiene cada cadena en Σ_n^*. Específicamente,

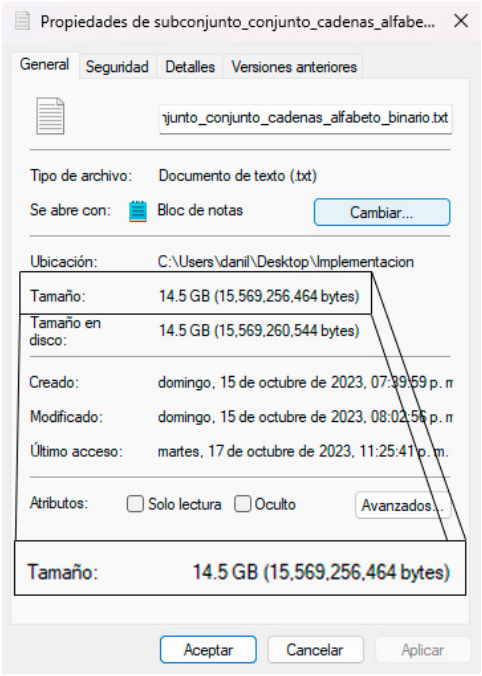


Figura 3.4: Propiedades del archivo de salida que contiene Σ_n^*.

la primer cantidad en el archivo tiene el número de apariciones del símbolo '1' en la primer cadena de sigma, la segunda cantidad tiene en número de apariciones del símbolo '1' en la segunda cadena de sigma, y así sucesivamente.

En la figura 3.5 se observa parte del contenido del archivo generado por el programa después de computar sigma con $n = 28$, para observar esta parte de contenido se necesitó codificar un programa que puediera abrir un flujo con el archivo, pues ningun software del sistema operativo Windows podia abrirlo (consideré el hecho de que ocupa 17680388553 bytes).

Este archivo es de gran importancia, pues el software que se utilizó para generar la gráfica visual es de alto nivel y por tanto tardaría mucho en procesar el archivo principal que contiene a sigma. La generación de la grafica se verá en la siguiente sección.

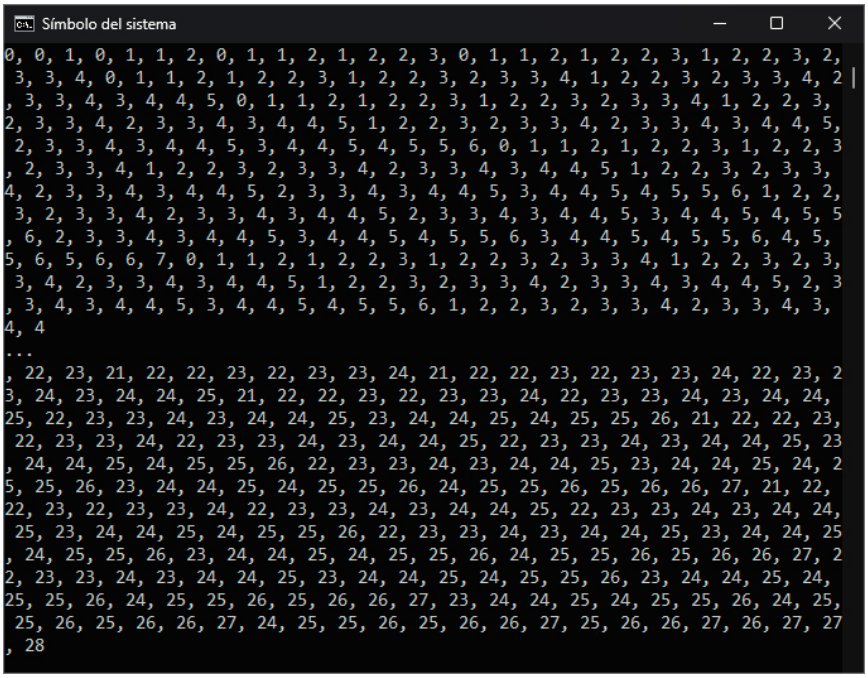


Figura 3.5: Valores del archivo CSV que representan el número de 1s de Σ_{28}^*.

3.2. Gráfica

Graficar el archivo CSV generado por el programa es una tarea dificultosa, pues al contar el número de puntos que se deben dibujar en la gráfica, tenemos que es una cantidad muy grande de RAM que deberá a MATLAB para que la pueda generar.

Número de puntos

Determinar el número de puntos es equivalente a determinar la cardinalidad de Σ_n^*. El número de elementos o la *cardinalidad* de un conjunto D se expresa como $|D|$, así, tenemos que

$$\begin{aligned} |\Sigma_n^*| &= |\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \cup \Sigma^{n-1} \cup \Sigma^n| \\ &= |\bigcup_{i=0}^{n} \Sigma^i| \\ &= \sum_{i=0}^{n} |\Sigma^i| \end{aligned}$$

Luego, por combinatoria sabemos que

$$|\Sigma^n| = |\Sigma|^n$$

Por consiguiente

$$\begin{aligned} |\Sigma_n^*| &= \sum_{i=0}^{n} |\Sigma^i| \\ &= \sum_{i=0}^{n} |\Sigma|^i \end{aligned}$$

Multiplicando ambos lados por $|\Sigma|$, se tiene

$$|\Sigma_n^*| = \sum_{i=0}^{n} |\Sigma^i| \Leftrightarrow |\Sigma||\Sigma_n^*| = |\Sigma| \sum_{i=0}^{n} |\Sigma|^i$$

Simplificando

$$\begin{aligned} |\Sigma||\Sigma_n^*| &= |\Sigma| \sum_{i=0}^{n} |\Sigma|^i \\ &= \sum_{i=0}^{n} |\Sigma||\Sigma|^i \\ &= \sum_{i=0}^{n} |\Sigma|^{i+1} \\ &= \sum_{j=1}^{n+1} |\Sigma|^j \\ &= \sum_{j=1}^{n} |\Sigma|^j + |\Sigma|^{n+1} \end{aligned}$$

Restando $|\Sigma_n^*|$ a ambos lados de la igualdad

$$\begin{aligned} |\Sigma||\Sigma_n^*| - |\Sigma_n^*| &= \sum_{j=1}^{n} |\Sigma|^j + |\Sigma|^{n+1} - \sum_{i=0}^{n} |\Sigma|^i \\ &= \sum_{j=1}^{n} |\Sigma|^j + |\Sigma|^{n+1} - \sum_{i=1}^{n} |\Sigma|^i - |\Sigma|^0 \\ &= |\Sigma|^{n+1} - |\Sigma|^0 \\ &= |\Sigma|^{n+1} - 1 \end{aligned}$$

Por consiguiente

$$|\Sigma_n^*| = \frac{|\Sigma|^{n+1} - 1}{|\Sigma| - 1} \tag{3.1}$$

Notese que $|\Sigma| \neq 0$ y $|\Sigma| \neq 1$, es decir, debemos tener más de un simbolo.

Graficando para $n = 28$

Como en este programa estamos considerando el alfabeto binario,tenemos que

$$|\Sigma_n| = |0, 1| = 2$$

Por consiguiente, usando (3.1), obtenemos

$$\begin{aligned} |\Sigma_n^*| &= \frac{|\Sigma|^{n+1} - 1}{|\Sigma| - 1} \\ &= \frac{2^{n+1} - 1}{2 - 1} \\ &= 2^{n+1} - 1 \end{aligned}$$

Ahora, si evaluamos para $n = 15$ obtenemos

$$|\Sigma_{15}^*| = 2^{15+1} - 1 = 65535$$

Que no es una cantidad muy grande de puntos y MATLAB podria manejar sin ningun problema en cualquier computadora. Pero, si evaluamos ahora para $n = 28$ obtendremos que

$$|\Sigma_{28}^*| = 2^{28+1} - 1 = 536870911$$

RAM necesaria. Si suponemos que MATLAB asigna a cada punto dos números enteros de 4 bytes, ocupará 4294967288 bytes de RAM graficar el archivo generado con $n = 28$.

Graficando "manualmente"

Ahora que se sabe que software como MATLAB, Wolfram Mathematica o incluso Python llenaría la RAM de cualquier computadora al generar la gráfica. Se deberá idear un metodo inteligente para graficar cada punto sin almacenarlo en la RAM, que esta comprobado que los softwares mensionados no lo hacen en sus funciones estandar para graficar.

El metodo para graficar sera "dibujar" cada punto directamente en una imagen que contendrá la gráfica completa. Por tanto se debera crear una

matriz bidimensional donde cada celda contendrá un valor booleano que representara un punto y en la imagen un pixel de color blanco. Como el valor maximo en eje x de la grafica, que representa la posición del elemento en Σ_n^*, es $|\Sigma_{28}^*| = 536870911$, la imagen tendría dimensiones gigantescas. Ante esto se aplicará una escala a este eje, que es lo que hacen todos los softwares para gráficas. Igualmente se aplicará una escala al eje y, que representa la cantidad de ceros que tiene el elemento en Σ_n^* que esta en la posición x.

Escalas de la gráfica

Si se quiere que el ancho de la imagen que contiene la gráfica sea de 10000 pixeles entonces a las abscisas de cada punto se les debe de multiplicar un factor tal que al abscisa máxima tenga el valor de 10000 pixeles. Es decir

$$|\Sigma_{28}^*| * 2 = m * 10000\ px$$

Donde m es el factor que "escala" a la gráfica. El factor 2 es porque abra un pixel del mismo color que el fondo, este pixel funcionará como serparador para una mejor visualicación de los puntos al hacer zoom desde un programa para visualizar imagenes.

Luego, de la ecuación anterior obtenemos que

$$m = \frac{10000}{|\Sigma_{28}^*|} = \frac{536870911 * 2}{10000} \approx 107374$$

Como cada punto representa un pixel en una imagen, las coordenas escaladas deben ser enteros, por esto se decidio redondear.

Para una apreciasión mejor de la imagen se decidió por que esta tuviera una escala 16:9 que es muy común en imagenes. Por lo tanto la altura de la imagen debe ser de

$$\frac{10000\ px \cdot 9}{16} = 5625\ px$$

Finalmente, se sabe que el número mayor de unos que tendrá cualquier elemento en Σ_n^x sera n. De este modo, para $n = 28$ el mayor número de unos que se tendrá es 28. Como acabamos de calcular que la altura de la imagen será de 5625 pixeles, entonces se debe multiplicar la ordenada de cada punto por el escalar

$$\frac{5625}{28} \approx 201$$

Script para graficar en MATLAB

Siguiendo la idea de guardar una matriz donde cada elemento representa un pixel en la imagen de la gráfica cuyos ejes se es escalan, se escribió el script del listing 3.2.

```
fileID = fopen("numero_de_unos.txt", 'r');
formatSpec = '%d,\t';

imagen = zeros(1, 1);

primero = 1;
ultimo = 0;
while feof(fileID) == 0
    A = fscanf(fileID, formatSpec, 10000);
    ultimo = ultimo + length(A);

    j = 1;
    for i = primero:ultimo
        imagen(A(j) * 201  + 1, 2 * round(i / 107374) + 1) = 1;
        j = j + 1;
    end

    clear A;

    primero=ultimo + 1;
end

imshow(imagen);
imwrite(imagen, 'grafica_puntos.png');
fclose(fileID);
```

Listing 3.2: Script de MATLAB para generar la imagen con la gráfica del número de 1s de Σ_n^x

Vale la pena observar que en vez de obtener número a número del archivo CSV, se usa un buffer de 10000 números. El buffer contribuye a que la ejecución del script sea más rápida pues hay un gasto considerable del procesador al obtener datos de archivos.

Imagen generada

Después de que se le dio la orden al programa para computar Σ_n^x y además de generar el archivo con el número de 1s de cada elemento de ese conjunto, llego el momento de ejecutar el script. La imagen que contiene la gráfica se observa en la figura 3.6. Se debe prestar atención a que la imagen fue rotada 90 grados en el sentido horario para una mejor visualización.

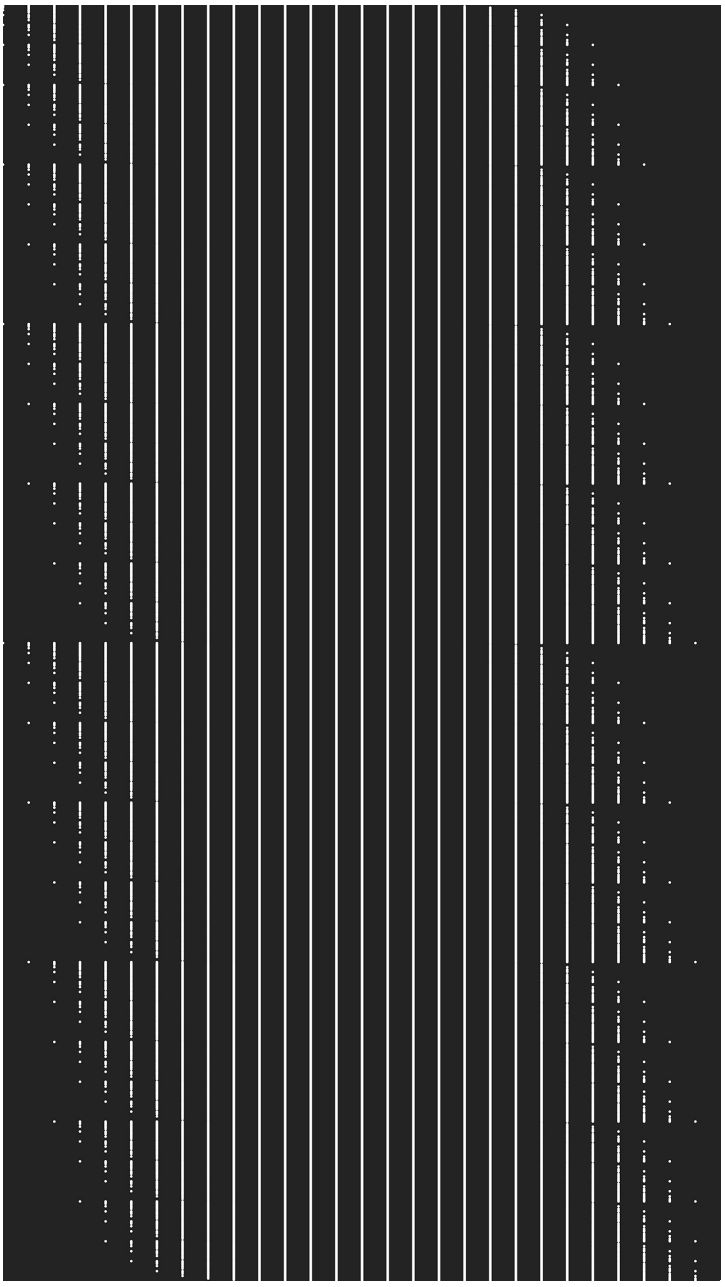


Figura 3.6: Imagen rotada de la gráfica que relaciona la posición de un elemento de Σ_n^* con el número de 1s presentes en el elemento.

3.3. El patrón

Observando la figura 3.6 sí se encuntra un patron, pero solo nos dice que el número de 1s de cada elemento de Σ_n^* sube y baja en diferentes rangos. Dados dos puntos consecutivos donde uno esta muy arriba y otro muy abajo, se sabe que principalmentes es cuando pasamos de un conjunto Σ^k a un Σ^{k+1}, con $k < n$. Tambien es cuando pasamos de una cadena de longitud l con $l-1$ 1s a una con un 1 y $l-1$ ceros.

El patrón que se observa en la grafica de la figura 3.6 se repetira no importa el valor de n con el que se basan los datos de esta.

Es impresionante ver como un problema que tal vez no tenga usos prácticos, involucre soluciones por lo menos dificultosas y permite comprender lo complejo que puede ser el software dedicado al análisis de grandes cantidades de datos. Tambien ofrece una clara respuesta al por qué los ingenieros, cientificos y matemáticos buscan algoritmos con complejidad polinomial.

A una comprensión competente del manejo de memoria y procesamiento a bajo nivel le sigue resultados coherentes y predecibles, sin esta comprensión la maquina obedecerá instrucciones que pueden producir eventos catastróficos para ella.

Bibliografía

[1] Francisco Javier Ceballos. *C/C++. Curso de programación*. 3.ª ed. RaMa, 2007.

[2] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. *Teoría de autómatas, lenguajes y computación*. 3.ª ed. Addison-Wesley, 2007.

[3] Ralph P. Grimaldi. *Matemáticas discreta y combinatoria. Una introducción con aplicaciones*. 3.ª ed. Addison-Wesley, 1994.

Apéndice A

Código en C completo del programa

A.1. universo.c

El listing A.1 muestra el codigo en univero.c, aquí se define la funcion main y funciones para manejar la entrada y salida en consola y para computar Σ_n^*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #include "alfabeto.h"
6  #include "cadena.h"
7
8  #define debug() printf("|E|")
9
10 // ------------------------------
11 // Funciones para la interfaz
12 int ImprimirInterfaz();
13 int ProcesarEntrada(char* entrada);
14 int ObtenerNroComando(char* entrada_sin_espacios);
15 int ConfirmarEntrada(char* mensaje);
16 int ValidarEntrada(int n, char* mensaje_invalida);
17
18 // ------------------------------
19 // Variables para la interfaz
20 char* comandos[5] = {"ayuda", "n=", "graficar", "salir"};
21
22 // ------------------------------
23 //Funciones que llaman los comandos desde la interfaz
```

```
24 int Ayuda();
25 int NIngresada(char* entrada);
26 int Graficar();
27
28 // ------------------------------
29 // Funciones para el algoritmo
30 int Computar(int n);
31 void Strings(int n, int dospotn);
32 void Guardar(int *N, int n);
33 int GenerarDatosGrafica();
34
35 // ------------------------------
36 // Variables para el algoritmo
37 int n_max = 1000;
38 int n_min = 0;
39 FILE* archivo_principal;
40 char nombre_archivo_principal[256] =
41     "subconjunto_conjunto_cadenas_alfabeto_binario.txt";
42
43 // ------------------------------
44 // ------------------------------
45 // ------------------------------
46 // Funcion principal
47
48 int main(int argc, char *argv[])
49 {
50     srand(time(NULL));
51
52     char* entrada;
53     int salir = 0;
54
55     ImprimirInterfaz();
56     putchar('\n');
57
58     while(!salir)
59     {
60         printf("\tEntrada:\t");
61
62         fflush(stdin);
63         entrada = LeerLinea(stdin, 1024, '\n');
64         fflush(stdin);
65
66         // No tiene por que imprimir un salto de linea al final
67         salir = ProcesarEntrada(entrada);
68         putchar('\n');
69
70         free(entrada);
71     }
72
```

```
73      return 0;
74  }
75
76  // ------------------------------
77  // ------------------------------
78  // ------------------------------
79  // Funciones para la interfaz
80
81  int ImprimirInterfaz()
82  {
83      FILE* archivo = fopen("interfaz.txt", "rt");
84      if(archivo == NULL)
85      {
86          printf("ERROR: No se abrio interfaz.txt");
87          return 1;
88      }
89
90      char caracter;
91      while(!feof(archivo))
92          if((caracter = fgetc(archivo)) != -1)
93              putchar(caracter);
94
95      fclose(archivo);
96      return 0;
97  }
98
99  int ProcesarEntrada(char* entrada)
100 {
101     char* entrada_no_espacios = EliminarEspacios(entrada);
102
103     int salir = 0;
104
105     switch (ObtenerNroComando(entrada_no_espacios))
106     {
107     case 0:
108
109         Ayuda();
110
111         break;
112     case 1:
113
114         NIngresada(entrada_no_espacios);
115
116         break;
117     case 2:
118
119         Graficar();
120
121         break;
```

```
122     case 3:
123 
124         salir = 1;
125 
126         break;
127     default:
128 
129         printf("Comando␣desconocido.");
130 
131         break;
132     }
133 
134     free(entrada_no_espacios);
135     return salir;
136 }
137 
138 int ObtenerNroComando(char* entrada_sin_espacios)
139 {
140     int nro;
141     if(EstaEn(entrada_sin_espacios, comandos[0]))
142         nro = 0;
143     else if(EstaEn(entrada_sin_espacios, comandos[1]))
144         nro = 1;
145     else if(EstaEn(entrada_sin_espacios, comandos[2]))
146         nro = 2;
147     else if(EstaEn(entrada_sin_espacios, comandos[3]))
148         nro = 3;
149     else
150         nro = 2147483648;
151 
152     return nro;
153 }
154 
155 int ConfirmarEntrada(char* mensaje)
156 {
157     puts(mensaje);
158     printf("\tConfirmar␣('1'␣->␣s");
159     putchar(195);
160     putchar(173);
161     printf("):\t");
162 
163     char confirmacion;
164     confirmacion = getchar();
165 
166     int r = 1;;
167     if(confirmacion != '1')
168         r = 0;
169 
170     return r;
```

```
171 }
172
173 int ValidarEntrada(int n, char* mensaje_invalida)
174 {
175     int r = 1;
176     if(!(n_min <= n && n <= n_max))
177     {
178         if(mensaje_invalida != NULL)
179             printf("%s", mensaje_invalida);
180         r = 0;
181     }
182     return r;
183 }
184
185 // ------------------------------
186 // ------------------------------
187 // ------------------------------
188 // Funciones que llaman los comandos desde la interfaz
189
190 int Ayuda()
191 {
192     FILE* archivo = fopen("ayuda.txt", "rt");
193     if(archivo == NULL)
194     {
195         printf("ERROR: No se abrio ayuda.txt");
196         return 1;
197     }
198
199     char caracter;
200     while(!feof(archivo))
201         if((caracter = fgetc(archivo)) != -1)
202             putchar(caracter);
203
204     fclose(archivo);
205     return 0;
206 }
207
208 int NIngresada(char* entrada)
209 {
210     int n;
211     if(EstaEn(entrada, "aleatorio"))
212         n = rand() % 1001;
213     else
214         n = atoi(entrada + BuscarC(entrada, '=') + 1);
215
216     int confirmado;
217     int valido = ValidarEntrada(n, "n invalido.\n");
218
219     char* memoria = NULL;
```

```
220     char* mensaje = NULL;
221     if(valido)
222     {
223         char* cadena[5];
224         cadena[0] = "Para␣n␣=␣";
225         char n_cadena[16];
226         cadena[1] = itoa(n, n_cadena, 10);
227         cadena[2] = ",␣el␣archivo␣ocupara␣";
228         cadena[3] = memoria = M(n);
229         cadena[4] = "␣byte␣de␣memoria.";
230         mensaje = Concatenar(cadena, 5);
231 
232         confirmado = ConfirmarEntrada(mensaje);
233     }
234 
235     if(!valido || !confirmado)
236         printf("Cancelado.");
237     else if(confirmado)
238     {
239         time_t t0, t;
240         printf("Tiempo:␣%lld␣s␣desde␣1␣de␣enero␣de␣1970",
241             t0 = time(NULL));
242 
243         printf("\nComputando...");
244         Computar(n);
245         printf("\nComputo␣termiando.");
246 
247         printf("\nTiempo:␣%lld␣s␣desde␣1␣de␣enero␣de␣1970",
248             t = time(NULL));
249         printf("\nDeltaT:␣%lld␣s", t - t0);
250     }
251 
252     free(memoria);
253     free(mensaje);
254 }
255 
256 int Graficar()
257 {
258     time_t t0, t;
259     printf("Tiempo:␣%lld␣s␣desde␣1␣de␣enero␣de␣1970",
260         t0 = time(NULL));
261 
262     printf("\nComputando␣datos...");
263     GenerarDatosGrafica();
264     printf("\nComputo␣terminado.");
265 
266     printf("\nTiempo:␣%lld␣s␣desde␣1␣de␣enero␣de␣1970",
267         t = time(NULL));
268     printf("\nDeltaT:␣%lld␣s", t - t0);
```

```
269 }
270
271 // ------------------------------
272 // ------------------------------
273 // ------------------------------
274 // Funciones para el algoritmo
275
276 int Computar(int n)
277 {
278     archivo_principal = fopen(nombre_archivo_principal, "wt");
279     if(archivo_principal == NULL)
280         return 1;
281
282     fprintf(archivo_principal, "{0,␣1}_n␣^␣*␣=␣{e");
283
284     for(int i = 1, dos_pot_i = 2; i <= n; i++, dos_pot_i *= 2)
285         Strings(i, dos_pot_i);
286
287     putc('}', archivo_principal);
288
289     fclose(archivo_principal);
290     return 0;
291 }
292
293 void Strings(int n, int dospotn)
294 {
295     int* N = (int*)malloc(sizeof(int) * n);
296     if(N == NULL)
297         return;
298     for(int i = 0; i < n; i++)
299         *(N + i) = 0;
300
301     Guardar(N, n);
302     for(int i = 1; i <= dospotn - 1; i++)
303     {
304         Sucesor(N, n);
305         Guardar(N, n);
306     }
307     free(N);
308 }
309
310 void Guardar(int *N, int n)
311 {
312     putc(',', archivo_principal);
313     putc('\t', archivo_principal);
314     for(int i = 0; i < n; i++)
315         putc(*(N + n - i - 1) + 48, archivo_principal);
316 }
317
```

```
318 int GenerarDatosGrafica()
319 {
320     FILE* archivo = fopen(nombre_archivo_principal, "rt");
321     if(archivo == NULL)
322         return 1;
323     FILE* datos = fopen("numero_de_unos.txt", "wt");
324     if(datos == NULL)
325     {
326         fclose(archivo);
327         return 1;
328     }
329 
330     char c;
331 
332     while(!feof(archivo) && getc(archivo) != 'e');
333     fputc('0', datos);
334     c = getc(archivo);
335 
336     char cadena[1000];
337     int cadena_n;
338     int cantidad_unos;
339     if(c != '}')
340         while(!feof(archivo))
341         {
342             cadena_n = 0;
343             while(!feof(archivo) && (c = getc(archivo)) != ',')
344             {
345                 cadena[cadena_n] = c;
346                 cadena_n++;
347             }
348             cadena[cadena_n] = '\0';
349 
350             cantidad_unos = ContarC(cadena, '1');
351 
352             fprintf(datos, ", %d", cantidad_unos);
353         }
354 
355     fclose(archivo);
356     fclose(datos);
357     return 0;
358 }
```

Listing A.1: Contenido de universo.c.

A.2. cadena.c

El listing A.2 (despues del codigo de cabecera) muestra el codigo en cadena.c, aquí se definen funciones para manejo de cadenas de caracteres.

```
#include <stdio.h>

int EstaEn(char *cadena, char *subcadena);
char* EliminarEspacios(char *cadena);
char* LeerLinea(FILE* archivo,
                int linea_longitud_max,
                char cfdl);
int BuscarC(char *cadena, char c);
int ContarC(char *cadena, char c);
char* Concatenar(char** cadena, int cadena_n);
```

```
#include <stdio.h>
#include <stdlib.h>

int EstaEn(char *cadena, char *subcadena)
{
    int esta = 0;

    int i = 0;
    int i_subcadena = 0;
    while(cadena[i] && subcadena[i_subcadena])
    {
        if(cadena[i] == subcadena[i_subcadena])
            i_subcadena++;
        else
            i_subcadena = 0;
        i++;
    }

    if(!subcadena[i_subcadena])
        esta = 1;
    else
        esta = 0;
    return esta;
}

char* EliminarEspacios(char *cadena)
{
    int n = 0;
    for(int i = 0; cadena[i]; i++)
        if(cadena[i] != ' ')
            n++;
    char* cadena_no_espacios =
        (char*)malloc(sizeof(char) * (n + 1));
```

```
34
35      for(int i = 0, j = 0; cadena[i]; i++)
36          if(cadena[i] != ' ')
37              cadena_no_espacios[j++] = cadena[i];
38      cadena_no_espacios[n] = 0;
39
40      return cadena_no_espacios;
41  }
42
43  char* LeerLinea(FILE* archivo, int linea_longitud_max,
44      char cfdl)
45  {
46      char* linea_max =
47          (char *)malloc(sizeof(char) * linea_longitud_max);
48      if(linea_max == NULL)
49          return NULL;
50
51      int n = 0;
52      char c;
53      while(n < linea_longitud_max &&
54          (c = getc(archivo)) != cfdl)
55      {
56          *(linea_max + n) = c;
57          n++;
58      }
59
60      char* linea = (char *)malloc(sizeof(char) * (n + 1));
61      if(linea == NULL)
62      {
63          free(linea_max);
64          return NULL;
65      }
66
67      for(int i = 0; i < n; i++)
68          *(linea + i) = *(linea_max + i);
69      *(linea + n) = '\0';
70
71      free(linea_max);
72      return linea;
73  }
74
75  int BuscarC(char *cadena, char c)
76  {
77      int indice;
78
79      int i = 0;
80      while(cadena[i] && cadena[i] != c)
81          i++;
82
```

```
83      if(!cadena[i])
84          indice = i - 1;
85      else
86          indice = i;
87
88      return indice;
89  }
90
91  int ContarC(char *cadena, char c)
92  {
93      int cantidad = 0;
94
95      int i = 0;
96      while(cadena[i])
97      {
98          if(cadena[i] == c)
99              cantidad++;
100         i++;
101     }
102
103     return cantidad;
104 }
105
106 char* Concatenar(char** cadena, int cadena_n)
107 {
108     char* conc;
109     int conc_n = 0;
110     int conc_longitud_final = 0;
111
112     char* actual;
113     int i, j;
114
115     for(i = 0; i < cadena_n; i++)
116     {
117         actual = cadena[i];
118         j = 0;
119         if(actual != NULL)
120             while(actual[j])
121             {
122                 conc_longitud_final++;
123                 j++;
124             }
125     }
126
127     conc = (char*)malloc(
128         sizeof(char) * (conc_longitud_final + 1));
129
130     for(i = 0; i < cadena_n; i++)
131     {
```

```
132         actual = cadena[i];
133         j = 0;
134         if(actual != NULL)
135             while(actual[j])
136             {
137                 conc[conc_n] = actual[j];
138                 conc_n++;
139                 j++;
140             }
141     }
142     conc[conc_n] = '\0';
143 
144     return conc;
145 }
```

Listing A.2: Contenido de cadena.c.

A.3. alfabeto.c

El listing A.3 muestra el codigo en alfabeto.c. Este código es la implementación de los algoritmos principales para el programa.

```
1  void Sucesor(int* N, int N_longitud);
2  void ImprimirN(FILE* archivo, int*, int);
3  char* M(int n);
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  
4  #include "logic.h"
5  
6  #include "./MiEntero/mientero.h"
7  
8  #include "alfabeto.h"
9  
10 void Sucesor(int* N, int N_longitud)
11 {
12     int conjuncion = 1;
13 
14     int previo = *(N);
15     *(N) = EXOR(*(N), conjuncion);
16     int i = 1;
17     while(conjuncion != 0 && i < N_longitud)
18     {
19         conjuncion = AND(conjuncion, previo);
20         previo = *(N + i);
21         *(N + i) = EXOR(*(N + i), conjuncion);
22         i++;
```

```
23      }
24  }
25
26  void ImprimirN(FILE* archivo, int* N, int N_longitud)
27  {
28      for(int i = 0; i < N_longitud; i++)
29          putc(*(N + N_longitud - i - 1) + 48, archivo);
30  }
31
32  char* M(int n)
33  {
34      ent resultado;
35      IniciarEnt(&resultado, "1");
36
37      ent aux;
38      IniciarEnt(&aux, "0");
39
40      for(int i = 0; i < n + 1; i++)
41      {
42          MultiplicaEntInt(resultado, 2, &aux);
43          CompiaEnt(aux, &resultado);
44      }
45
46      MultiplicaEntInt(resultado, n + 1, &aux);
47      CompiaEnt(aux, &resultado);
48
49      ent sum16;
50      IniciarEnt(&sum16, "16");
51      SumaEnt(resultado, sum16, &aux);
52      CompiaEnt(aux, &resultado);
53
54      char* resultado_cadena = EntACad(resultado);
55
56      LiberarEnt(&resultado);
57      LiberarEnt(&aux);
58      LiberarEnt(&sum16);
59      return resultado_cadena;
60  }
```

Listing A.3: Contenido de alfabeto.c.

A.4. logic.c

El listing A.4 muestra el codigo en logic.c, donde se definen funciones lógicas.

```
1  typedef int boolean;
2
```

```
3  #define true 1
4  #define false 0
5
6  #define NOT !
7
8  boolean AND(boolean p, boolean q);
9  boolean OR(boolean p, boolean q);
10
11
12 boolean THEN(boolean p, boolean q);
13 boolean EXOR(boolean p, boolean q);
```

```
1  #include "logic.h"
2
3  boolean AND(boolean p, boolean q)
4  {
5      boolean l = p * q;
6
7      return !!l;
8  }
9
10 boolean OR(boolean p, boolean q)
11 {
12     boolean l = p + q;
13
14     return !!l;
15 }
16
17
18 boolean THEN(boolean p, boolean q)
19 {
20     boolean l = OR(NOT p, q);
21
22     return l;
23 }
24
25 boolean EXOR(boolean p, boolean q)
26 {
27     boolean l = NOT (AND(THEN(p, q), THEN(q, p)));
28
29     return l;
30 }
```

Listing A.4: Contenido de logic.c.

A.5. mientero.c

El listing A.5 (desues del codigo de cabecera) muestra el codigo en mientero.c, esta libreria fue programada por motivos personales e ideas obtenidas gracias a la asignatura fundamentos de programación. La libreria proporciona una estructura de datos robusta para números grandes (con miles de millones de dígitos).

```
1  /* Manejo de enteros con digitos infinitos. */
2
3  #define MMAX 999999999
4
5  typedef struct VectorEntero
6  {
7      int *magnitud;
8      int magnitud_long;
9      short signo;    // 0 negativo, 1 positivo
10 } ent;
11
12 ent *SumaEnt(ent x, ent y, ent *destino);
13 ent *RestaEnt(ent minuendo, ent sustraendo, ent *destino);
14 ent *MultiplicaEntInt(ent vector, int escalar, ent *destino);
15 ent *MultiplicaEntEnt(ent factor1, ent factor2, ent *destino);
16 ent *CompiaEnt(ent origen, ent *destino);
17 int EsMayorAbsEnt(ent x, ent y);
18 void IniciarEnt(ent *x, char *contenido);
19 int ElimCerosNoSigEnt(ent *x);
20 void ImpEnt(ent x);
21 void LiberarEnt(ent *x);
22
23 char* EntACad(ent x);
```

```
1  /* Manejo de enteros con digitos infinitos. */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  #include "mientero.h"
7  #include "mimath.h"
8  #include "micadena.h"
9
10 ent *SumaEnt(ent, ent, ent *);
11 ent *RestaEnt(ent, ent, ent *);
12 ent *MultiplicaEntInt(ent, int, ent *);
13 ent *MultiplicaEntEnt(ent, ent, ent *);
14 ent *CompiaEnt(ent origen, ent *);
15 int EsMayorAbsEnt(ent, ent);
16 void IniciarEnt(ent *, char *);
```

```
17 int ElimCerosNoSigEnt(ent *);
18 void ImpEnt(ent);
19 void LiberarEnt(ent *);
20
21 ent *SumaEnt(ent x, ent y, ent *destino)
22 {
23     int c = EsMayorAbsEnt(y, x);  // |y| > |x|
24     if((x.signo && !y.signo) || (y.signo && !x.signo))
25     {
26         if(c)
27             RestaEnt(y, x, destino);
28         else
29             RestaEnt(x, y, destino);
30
31         destino->signo = x.signo && !y.signo ? !c : c;
32
33         return destino;
34     }
35
36     LiberarEnt(destino);
37     destino->signo = x.signo;
38
39     int sumando1, sumando2;
40     int sumaTotal, sobrante = 0;
41
42     int magMax = MaxInt(x.magnitud_long, y.magnitud_long);
43     for(int i = 0; i < magMax; i++)
44     {
45         sumando1 = i < x.magnitud_long ? x.magnitud[i] : 0;
46         sumando2 = i < y.magnitud_long ? y.magnitud[i] : 0;
47         sumaTotal = sumando1 + sumando2 + sobrante;
48
49         destino->magnitud = (int *)realloc(destino->magnitud,
50             sizeof(int) * (destino->magnitud_long + 1));
51         destino->magnitud_long += 1;
52
53         sobrante = (sumaTotal > MMAX);
54         destino->magnitud[destino->magnitud_long - 1] =
55             sumaTotal - (MMAX + 1) * sobrante;
56     }
57
58     if(sobrante)
59     {
60         destino->magnitud = (int *)realloc(destino->magnitud,
61             sizeof(int) * (destino->magnitud_long + 1));
62         destino->magnitud_long += 1;
63
64         destino->magnitud[destino->magnitud_long - 1] = 1;
65         sobrante = 0;
```

```
66      }
67
68      return destino;
69  }
70
71  ent *RestaEnt(ent minuendo, ent sustraendo, ent *destino)
72  {
73      LiberarEnt(destino);
74
75      int minParc, susParc;
76      int resta, accarreo = 0;
77
78      for(int i = 0; i < minuendo.magnitud_long; i++)
79      {
80          minParc = minuendo.magnitud[i];
81          susParc = i < sustraendo.magnitud_long ?
82              sustraendo.magnitud[i] : 0;
83          resta = minParc - (susParc + accarreo);
84
85          destino->magnitud = (int *)realloc(destino->magnitud,
86              sizeof(int) * (destino->magnitud_long + 1));
87          destino->magnitud_long += 1;
88
89          accarreo = (resta < 0);
90          destino->magnitud[destino->magnitud_long - 1] =
91              resta + (MMAX + 1) * accarreo;
92      }
93
94      ElimCerosNoSigEnt(destino);
95
96      return destino;
97  }
98
99  ent *MultiplicaEntInt(ent vector, int escalar, ent *destino)
100 {
101     LiberarEnt(destino);
102     destino->signo = (!vector.signo || EsPosInt(escalar)) &&
103         (!EsPosInt(escalar) || vector.signo);
104
105     long long r = AbsInt(escalar), multiplicacion;
106     int multMod, multDiv = 0;
107
108     for(int i = 0; i < vector.magnitud_long; i++)
109     {
110         multiplicacion = r * vector.magnitud[i] + multDiv;
111         multMod = multiplicacion % (MMAX + 1);
112         multDiv = multiplicacion / (MMAX + 1);
113
114         destino->magnitud = (int *)realloc(destino->magnitud,
```

```
115             sizeof(int) * (destino->magnitud_long + 1));
116         destino->magnitud_long += 1;
117 
118         destino->magnitud[destino->magnitud_long - 1] =
119             multMod;
120     }
121 
122     if(multDiv)
123     {
124         destino->magnitud = (int *)realloc(destino->magnitud,
125             sizeof(int) * (destino->magnitud_long + 1));
126         destino->magnitud_long += 1;
127 
128         destino->magnitud[destino->magnitud_long - 1] =
129             multDiv;
130         multDiv = 0;
131     }
132 
133     ElimCerosNoSigEnt(destino);
134 
135     return destino;
136 }
137 
138 ent *MultiplicaEntEnt(ent factor1, ent factor2, ent *destino)
139 {
140     LiberarEnt(destino);
141     IniciarEnt(destino, "0");
142 
143     short signo = (!factor1.signo || factor2.signo) &&
144         (!factor2.signo || factor1.signo);
145     factor1.signo = 1;
146     factor2.signo = 1;
147 
148     int facParc;
149     ent mulParc, aux;
150     IniciarEnt(&mulParc, "0");
151     IniciarEnt(&aux, "0");
152 
153     for(int i = 0; i < factor1.magnitud_long; i++)
154     {
155         facParc = factor1.magnitud[i];
156         MultiplicaEntInt(factor2, facParc, &mulParc);
157 
158         for(int j = 0; j < i; j++)
159         {
160             MultiplicaEntInt(mulParc, 1000000000, &aux);
161             CompiaEnt(aux, &mulParc);
162         }
163 
```

```
164         SumaEnt(*destino, mulParc, &aux);
165         CompiaEnt(aux, destino);
166     }
167 
168     destino->signo = signo;
169 
170     free(mulParc.magnitud);
171     free(aux.magnitud);
172 
173     return destino;
174 }
175 
176 ent *CompiaEnt(ent origen, ent *destino)
177 {
178     free(destino->magnitud);
179     destino->magnitud =
180         (int *)malloc(sizeof(int) * origen.magnitud_long);
181     destino->magnitud_long = origen.magnitud_long;
182     destino->signo = origen.signo;
183 
184     for(int i = 0; i < origen.magnitud_long; i++)
185         destino->magnitud[i] = origen.magnitud[i];
186 
187     return destino;
188 }
189 
190 int EsMayorAbsEnt(ent x, ent y) // |x| > |y|
191 {
192     int operacion = 0;
193 
194     if(x.magnitud_long == y.magnitud_long)
195     {
196         int i = 0;
197         while(i < x.magnitud_long &&
198             (x.magnitud[x.magnitud_long - 1 - i] ==
199                 y.magnitud[y.magnitud_long - 1 - i]))
200             i++;
201         if(i < x.magnitud_long &&
202             (x.magnitud[x.magnitud_long - 1 - i] >
203                 y.magnitud[y.magnitud_long - 1 - i]))
204             operacion = 1;
205     }
206     else
207         operacion = x.magnitud_long > y.magnitud_long;
208 
209     return operacion;
210 }
211 
212 void IniciarEnt(ent *x, char *contenido)
```

```
213 {
214     short signo = !(contenido[0] == '-');
215     int inicio = !signo;
216 
217     int contenido_long = 0;
218     while(contenido[inicio + contenido_long++]);
219     contenido_long--;
220     int magnitudes = CeilInt2(contenido_long, 9);
221 
222     x->magnitud = (int *)malloc(sizeof(int) * magnitudes);
223     x->magnitud_long = magnitudes;
224     for(int i = 0; i < x->magnitud_long; i++)
225         x->magnitud[i] = 0;
226     x->signo = signo;
227 
228     for(int i = 0, j = 0, potencia10 = 1;
229         j < contenido_long;
230             j++, i += (!(j % 9)) ? 1 : 0,
231             potencia10 = (!(j % 9)) ? 1 : (potencia10 * 10))
232         x->magnitud[i] += CharAInt
233             (contenido[inicio +
234                 contenido_long - 1 - j]) * potencia10;
235 
236     ElimCerosNoSigEnt(x);
237 }
238 
239 int ElimCerosNoSigEnt(ent *x)
240 {
241     int ceros = 0;
242     while(ceros < x->magnitud_long - 1 &&
243         x->magnitud[x->magnitud_long - 1 - ceros] == 0)
244         ceros++;
245 
246     ent xNuevo;
247     xNuevo.magnitud =
248         (int *)malloc(sizeof(int) * (x->magnitud_long -
249             ceros));
250     xNuevo.magnitud_long = x->magnitud_long - ceros;
251     for(int i = 0; i < xNuevo.magnitud_long; i++)
252         xNuevo.magnitud[i] = x->magnitud[i];
253     xNuevo.signo = x->signo;
254 
255     CompiaEnt(xNuevo, x);
256     free(xNuevo.magnitud);
257     return ceros;
258 }
259 
260 void ImpEnt(ent x)
261 {
```

```
262     if(!x.signo)
263         putchar('-');
264     printf("%d", x.magnitud[x.magnitud_long - 1]);
265     for(int i = 1; i < x.magnitud_long; i++)
266         printf("%09d", x.magnitud[x.magnitud_long - 1 - i]);
267 }
268
269 void LiberarEnt(ent *x)
270 {
271     free(x->magnitud);
272     x->magnitud = NULL;
273     x->magnitud_long = 0;
274     x->signo = 1;
275 }
276
277 char* EntACad(ent x)
278 {
279     char* cad =
280         (char*)malloc(sizeof(char) * (x.magnitud_long * 10));
281     if(cad == NULL)
282         return NULL;
283     long long int cad_n = 0;
284
285     if(!x.signo)
286         cad[cad_n++] = '-';
287
288     char magnitud_actual[16];
289     int i_mag_act;
290     for(int i = 0; i < x.magnitud_long; i++)
291     {
292         itoa(x.magnitud[x.magnitud_long - 1 - i],
293             magnitud_actual, 10);
294         i_mag_act = 0;
295         while(magnitud_actual[i_mag_act])
296         {
297             cad[cad_n] = magnitud_actual[i_mag_act];
298             cad_n++;
299             i_mag_act++;
300         }
301     }
302     cad[cad_n] = '\0';
303
304     return cad;
305 }
```

Listing A.5: Contenido de mientero.c.

A.6. mimath.c

El listing A.6 muestra el codigo en mimath.c, contiene funciones matemáticas básicas.

```
1  int MinInt(int x, int y);
2  int MaxInt(int x, int y);
3  int CeilInt2(int dividendo, int divisor);
4  int EsPosInt(int x);
5  int AbsInt(int x);
```

```
1  #include "mimath.h"
2
3  int MinInt(int x, int y)
4  {
5      if(x < y)
6          return x;
7      return y;
8  }
9
10 int MaxInt(int x, int y)
11 {
12     if(x > y)
13         return x;
14     return y;
15 }
16
17 int CeilInt2(int dividendo, int divisor)
18 {
19     int divicionCeil = dividendo / divisor;
20     if(dividendo % divisor)
21         divicionCeil += 1;
22
23     return divicionCeil;
24 }
25
26 int EsPosInt(int x)
27 {
28     if(x < 0)
29         return 0;
30     return 1;
31 }
32
33 int AbsInt(int x)
34 {
35     if(x < 0)
36         return -x;
37     return x;
```

```
38 }
```

Listing A.6: Contenido de mimath.c.

A.7. micadena.c

El listing A.7 muestra el codigo en micadenac.c. Por estructura de las librerias, unicamente contiene una función, esta funcion convierte un caracter a un entero.

```
1 int CharAInt(char c);
```

```
1 #include "micadena.h"
2 int CharAInt(char c){int digito = c - 48; return digito;}
```

Listing A.7: Contenido de micadena.c.