Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Algoritmos avanzados

Laboratorio 1 – Enumeración exhaustiva

Christofer Rodríguez

Profesor: Cristián Sepúlveda S.

Ayudante: César Rivera M.

Santiago – Chile
1-2021

Tabla de contenidos

Índice de figuras

Índice de tablas

1. INTRODUCCIÓN

La seguridad informática ha sido unos de los temas más importantes por mucho tiempo, con el fin de fortalecer la seguridad del cifrado y descifrado de claves Kasahara y Murakami propusieron una variación al antiguo esquema, capaz de proteger estas claves de distintos tipos de ataques. Esta variación del esquema consiste en un conjunto de números enteros con una ponderación correspondiente, con esto en mente es necesario implementar un programa en el lenguaje de programación C, que haciendo uso de algoritmos de enumeración exhaustiva, se capaz de determinar la mayor suma que se puede obtener de un subconjunto de números que a su vez no superen un límite en la suma total de sus ponderaciones.

Los objetivos de este informe son explicar el método utilizado para resolver este problema, describir los experimentos realizados para la obtención de datos, para finalmente analizar los resultados obtenidos, identificando los principales hallazgos y limitaciones de la solución propuesta.

Esto se llevará a cabo, primeramente, describiendo la metodología utilizada, explicando en que consiste el tipo de algoritmo utilizado, el proceso de implementación y los resultados obtenidos en los distintos experimentos realizados; luego se analizarán los resultados obtenidos en la sección de discusión, en esta sección se expondrán los principales hallazgos y falencias de la implementación, finalmente sugerirán mejoras para futuras investigaciones e implementaciones.

2. MÉTODO

Para comprender la implementación realizada, es necesario primero comprender en que consiste un algoritmo de enumeración exhaustiva; este tipo de algoritmo se basa en "generar" todas las posibles combinaciones del espacio de soluciones, a la vez que se comprueba que dichas combinaciones cumplan cierta propiedad. De manera general se busca generar una lista con todas las posibles soluciones, luego se comprueba que estas posibles soluciones cumplan la propiedad establecida, para finalmente, dependiendo del tipo de problema, entregar la lista de soluciones que cumplen esta propiedad o la mejor solución entre todas. Debido a la magnitud del espacio de soluciones, esta metodología es costosa en tiempo y memoria, producto de lo anterior, es considerada una metodología generalmente ineficiente.

Para solucionar este problema con la metodología de enumeración exhaustiva en el lenguaje de programación C, se desarrollaron 2 implementaciones:

2.1. Solución 1

Para esta implementación se decidió hacer uso de estructuras para representar los elementos con sus respectivos valores y ponderaciones, además de una estructura para representar una combinación con la suma de valores, suma de ponderaciones, posición de sus elementos y la cantidad de elementos que componían dicha combinación.

Ya que los datos para este problema son ingresados mediante un archivo de texto plano con cierta estructura, se implementó un algoritmo que, tras recibir el nombre del archivo de entrada, abre dicho archivo en modo lectura, y a la vez que este era leído, la información contenida en él se almacena en la estructura que representa un elemento. Luego de generar el elemento con la información correspondiente, este se agrega a un arreglo que, tras haber leído la totalidad del archivo, es retornado con todos los elementos en él.

Una vez leído el archivo de entrada, se procede a generar todas las posibles combinaciones de solo dos elementos y a almacenarlas en un arreglo, para esto primero es comprobado si esta posible combinación cumple con la condición de no superar el límite de ponderación total y ser única. Luego de generar todas las combinaciones compuestas por dos elementos, mediante dos ciclos anidados, se recorre el arreglo de combinaciones generadas (inicialmente solo con parejas de elementos) y un nuevo elemento era agregado en cada ciclo de este bucle interior; al igual que al formar las parejas, primero se comprueba si esta posible combinación no supera el límite de ponderación total, es una combinación única y no posee un elemento repetido.

Para conocer cuál es la mayor suma de valores posible a generar, tanto al momento de crear los elementos al leer el archivo de entrada, como al generar todo el resto de las combinaciones, se comparaba la suma total de valores y con la mayor suma de valores encontrada hasta el momento.

Una de las dificultades presentes en la realización de está solución fue el manejo de memoria, debido a que se decidió almacenar cada solución generada a lo largo del proceso y sumado a la gran cantidad de combinaciones que es posible obtener de este problema, fue necesario trabajar con arreglos dinámicos para almacenar las combinaciones generadas, de esta manera cada vez que una nueva soluciones es generada, se asigna la memoria exacta para almacenar las antiguas soluciones, más la nueva y estás eran almacenadas en el nuevo arreglo.

El código de esta solución fue analizado contando la cantidad de instrucciones a ejecutar para obtener el tiempo de ejecución teórico y el orden de complejidad respectivamente:

- $T(n) = 19 + 12n + 13(n^2) + (n^2) * ((n * (n-1))/2) + (n^3) + (2^n) * (n^2 + 2n) + (2^n) * (7n + 2(n^2)) + (2^n) * (n) * (3((2^n) * (n^2)) + n * (2^n) + 3(2^n)) + (2^n) * (n) * (4 + 2((2^n) - 1)) + 2((2^n) * (n))$.

- $O(2^n)$.

La ecuación del orden de complejidad se utilizó para representar el aumento en el tiempo de ejecución del algoritmo en el siguiente gráfico:

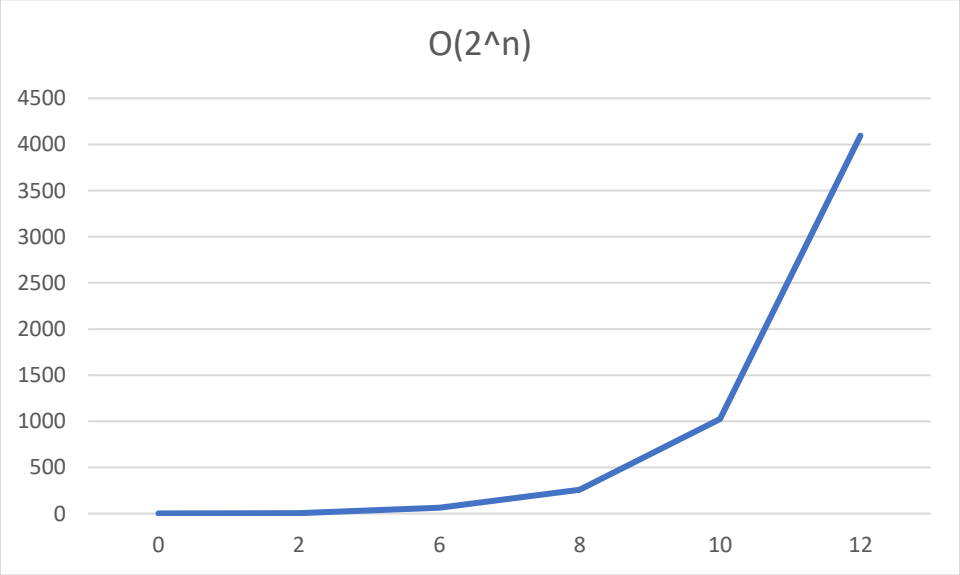

Ilustración 1: Gráfico de la complejidad algorítmica con entradas pequeñas

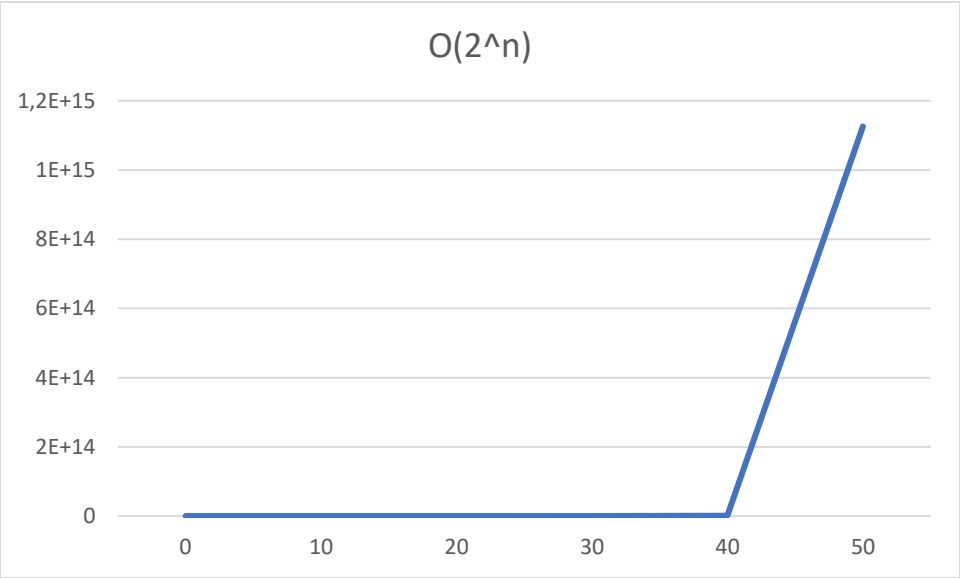

Ilustración 2: Gráfico de la complejidad algorítmica con entradas grandes

Además, para contrastar el tiempo teórico con el tiempo real de ejecución, se hizo uso de la librería **time.h** para la toma del tiempo de ejecución del programa con las entradas. Es importante indicar que solamente se pudo realizar 2 experimentos con la ejecución de esta solución, ya que esta solución era inviable.

Tabla 1: Tamaño entrada vs tiempo de ejecución, solución 1

Tamaño de entrada	Tiempo de ejecución (seg)
10	1
20	3600+

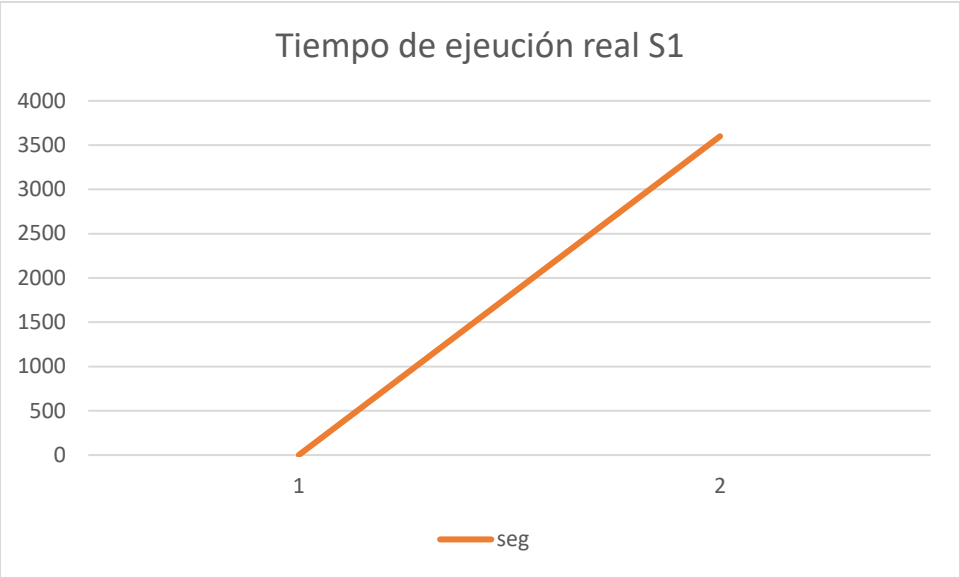

Ilustración 3: Tiempo de ejecución real, solución 1

2.2. Solución 2

Debido a que no se logró resolver el problema mediante la primera solución creada, fue necesario implementar una nueva solución para resolver el problema. Se mantuvo la estructura para almacenar los elementos del archivo de entrada, además del algoritmo para la lectura y almacenaje de estos elementos en un arreglo.

A diferencia de la primera solución, se decidió no utilizar la estructura para representar las combinaciones con sus componentes, en cambio se decidió utilizar un arreglo de tipo **char** para representar un número binario con una cantidad de bits igual a la cantidad de elementos presentes en el archivo de entrada, esto ya que la cantidad de números binarios que se pueden representar con cierta cantidad de bits, es igual a la cantidad de combinaciones sin repetición que se pueden generar con cierta cantidad de elementos y esta está dada por la expresión 2^n.

Para "generar" todas las posibles combinaciones , se hizo uso de solo un ciclo **for** que se ejecuta una cantidad de veces dada por la expresión 2^n, que como se explicó anteriormente, corresponde a la cantidad de posibles combinaciones a generar, donde "n" corresponde a la cantidad de elementos en el archivo de entrada. En cada ciclo ejecutado, se suma el número binario 1 al número binario contenido en el arreglo, una vez sumado, se recorre dicho numero binario en donde el número de bit revisado corresponde a la posición de un elemento dentro del arreglo de elementos, si el número en dicho bit es un 1, se considera dicho elemento para la combinación, por el contrario, si es no 0 no se considera. A medida que se comprueba cada bit del numero binario, se suma el valor de los elementos y su ponderación, si mientras se está realizando la comprobación de los bits, se supera la suma de ponderación límite, se descarta esta posible solución.

Al igual que en la primera solución, al momento de almacenar los elementos, como al generar las soluciones, se compara la mayor suma de valores encontrada con la nueva suma, si está es mayor, es almacenada como la nueva mayor suma encontrada.

Como también se realizó para la solución 1, se analizo la solución 2, contando la cantidad de instrucciones a ejecutar en el código fuente para llegar a las siguientes expresiones que representan el tiempo de ejecución teórico y el orden de complejidad respectivamente.

- $T(n) = 17 + 13n + 2^n + 2^n * (6 + 3n)$.
- $O(2^n)$.

Se decidió no incluir un gráfico para el tiempo de ejecución teórico para esta solución, ya que al compartir el mismo orden de complejidad con la solución 1, los gráficos representan el comportamiento teórico de ambas soluciones.(Ilustración 1 e Ilustración 2)

Además, para contrastar el tiempo teórico con el tiempo real de ejecución, se hizo uso de la librería **time.h** para la toma del tiempo de ejecución del programa con las diferentes entradas:

Tabla 2: Tamaño entrada vs tiempo de ejecución, solución 1

Tamaño de entrada	**Tiempo de ejecución (seg)**
10	0
20	0,225
30	138,716
40	3600+

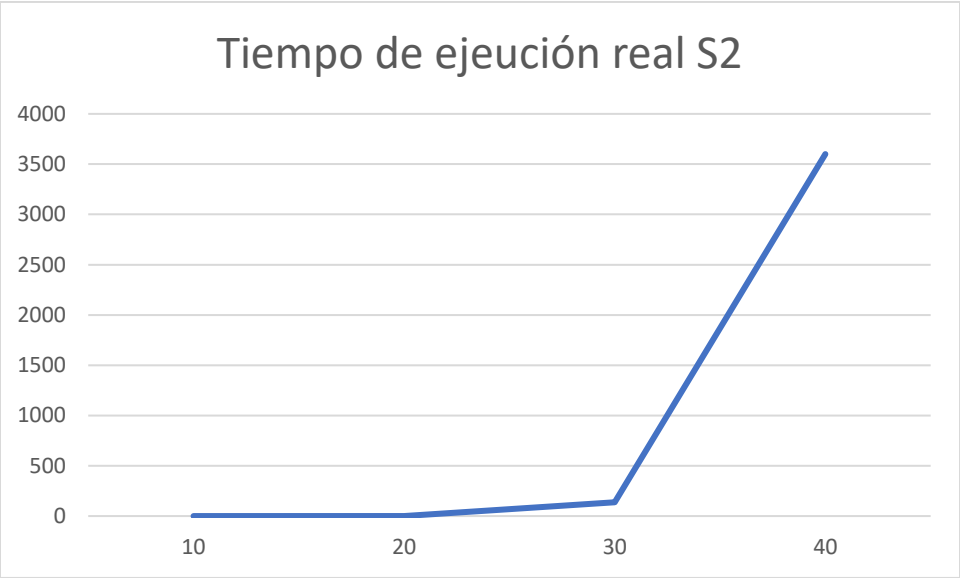

Ilustración 4: Tiempo de ejecución real, solución 2

(*) Es importante señalar que los experimentos realizados con ambos programas se realizaron en un computador común con sistema operativo Windows, con un tiempo máximo de 1 hora para cada experimento. Los resultados pueden variar en un computador, entorno o situación diferente.

3. DISCUSIÓN

3.1. Análisis solución 1

Luego de analizar los resultados obtenidos de los experimentos realizados a la primera solución planteada, se puede observar que claramente esta solución no es viable para este problema. Como podemos observar en el gráfico de la Ilustración 1, se esperaba que incluso con un pequeño aumento en el tamaño de la entrada, la cantidad de tiempo de ejecución se elevara bastante; pero los resultados obtenidos fueron peor que lo esperado, en la Tabla 1 podemos ver que para una entrada de 10, la solución al problema es entregada de manera correcta casi de manera inmediata, pero al incrementar el tamaño de la entrada a 20, el programa no fue capaz de terminar su ejecución en el tiempo máximo de prueba (1 hora).

Las falencias de esta solución son múltiples, pero principalmente radica en el poco análisis del problema previo al desarrollo de una implementación. Desde un comienzo se decidió utilizar estructuras para representar tanto los elementos como las combinaciones y almacenar cada uno de estos "componentes" en sus correspondientes arreglos; debido a que en esta implementación se utilizaron las combinaciones generadas para la construcción de las futuras combinaciones, el manejo de memoria debía ser un punto importante para considerar, por

esto se trabajó con arreglos dinámicos, asignando solo la memoria a utilizar por las actuales combinaciones.

Si bien la decisión anteriormente descrita fue una buena "solución parcial", intercambió uso de memoria por cantidad de instrucciones a realizar, ya que cada vez que se creaba una nueva combinación se debía traspasar todas las soluciones al nuevo arreglo. Sumado a esto debido a la forma en la que se generaban las nuevas soluciones, específicamente la selección de elementos que compondrían dicha combinación era posible generar combinaciones que previamente ya fueron creadas, por esto era necesario comprobar si esta solución era única o si el nuevo elemento a agregar no se encontraba ya dentro de la combinación. Teniendo en cuenta que se debía generar todas las posibles combinaciones y que el tamaño crecía rápidamente con un pequeño cambio en el tamaño de la entrada, esta implementación solamente agregaba más "trabajo" al momento de generar cada una de estas combinaciones, por lo que esta solución no fue capaz de entregar una respuesta a una entrada de tamaño 20 dentro del tiempo establecido para el experimento.

Una mejora a esta implementación podría ser el utilizar una estructura de datos diferente para almacenar la información, por ejemplo, listas enlazadas, pero aun cuando este cambio permitiría resolver el problema de trasladar de arreglo todas las combinaciones una vez encontradas una nueva, a pesar de esta mejora, este tipo de estructura haría más difícil el acceso a un elemento en específico almacenado en él. Como se nombró en un principio, la principal falencia fue el "pobre" análisis previo al desarrollo de la solución, ya que desde un comienzo no era necesario guardar cada combinación generado, solamente la mayor suma total de valores encontrada.

Con este pensamiento en mente, se desarrollo la solución 2 que será analizada a continuación:

3.2. Análisis solución 2

Como se puede observar en la Tabla 2, esta solución presentó mucho mejores resultados que la anterior solución, para una entrada de tamaño 20, con la cual la primera solución no pudo entregar una solución en el tiempo máximo, esta era capaz de entregar una respuesta casi inmediata.

Los principales puntos a destacar de esta implementación en comparación a la anterior, es que, al no almacenar las combinaciones, no es necesario crear estas estructuras y construir un lugar donde guardarlas, además al generar las combinaciones utilizando una representación de un número binario, dejó de ser necesario el comprobar si la nueva combinación era única o si el elemento agregado no se encontraba incluido de la combinación, esto disminuyó enormemente la cantidad de instrucciones a realizar por el programa, esto se puede ser reflejado en la expresión del tiempo de ejecución teórico (Revisar sección 2.2) y de una manera más visual en la Ilustración 4.

A pesar de las mejoras realizadas, una de las limitaciones del programa es que no fue capaz de entregar una respuesta para una entrada de tamaño 40 dentro del tiempo máximo para las

pruebas (1 hora). Esta limitación puede ser causada por la forma en la que se crea la representación del número binario, luego de leer el archivo de entrada y conocer la cantidad de elementos presentes en este, es generado un número binario con una cantidad de bits igual a la cantidad de elementos. Lo anterior provoca que, para entradas de tamaño elevado, por ejemplo 40, al comienzo de la generación de las combinaciones se deberá revisar 40 posiciones de este arreglo que representa en número binario para saber qué elementos componen la combinación y en las "primeras" combinaciones donde los números representados de manera binario solo utilizan 1, 2 o 3 bits, igualmente será necesario revisar el resto de las posiciones, realizando una gran cantidad de instrucciones innecesarias en cada ciclo.

Una manera de mejorar esta falencia sería el generar una representación del número binario con solo la cantidad de bits para representar el número del ciclo actual (número que representa la combinación) y cuando se requieran más bits para representar la siguiente combinación, se genera un número binario un bits más, ya que la cantidad de elementos coincide con la cantidad de bits necesario para realizar todas las combinaciones, se deberá modificar la cantidad de bits del número binario n veces, siendo n el tamaño de la entrada.

4. CONCLUSIÓN

A pesar de que no se logró obtener una respuesta para todos los archivos de entrada de diferentes tamaños, gracias a los datos obtenidos mediante los diversos experimentos y al análisis de estos, si se logró comprobar que el comportamiento de la curva que representa el tiempo de ejecución real frente a diferentes tamaños de entrada, se asemejaba bastante a la curva de la complejidad algorítmica obtenida del tiempo de ejecución teórico, además que se consiguió conocer las ventajas y limitaciones de algoritmos de enumeración exhaustiva. Como principal aprendizaje se puede destacar el apropiado análisis tanto del problema, como de la solución a implementar previo al desarrollo de esta, junto con junto con la constante consideración de la cantidad de instrucciones a ejecutar y la memoria utilizada. Todo lo aprendido y obtenido en la realización de este laboratorio, será beneficioso para la realización de futuros laboratorios o implementaciones a desarrollar.

5. APENDICE

Pseudo código solución 2:

```
Estructura Elemento
      num valor
      num ponderacion

Elemento crearElemento(num valorNuevo, num ponderacionNuevo)
      Elemento nuevoElemento
      nuevoElemento.valor = valorNuevo
      nuevoElemento.ponderacion = ponderacionNuevo
      RETURN nuevoElemento

Void aumentarBinario(Array caracter numeroBinario, num largo)
      IF numeroBinario[largo-1] = '0' THEN
            numeroBinario[largo-1] = '1'
            RETURN

      ELSE IF numeroBinario[largo-1] = '1' THEN
            numeroBinario[largo-1] = '0'
            FOR num i=largo-2 TO 0
                  IF numeroBinario[i] = '0' THEN
                        numeroBinario[i] = '1'
                        RETURN

                  ELSE
                        numeroBinario[i] = '0'

                  i = i - 1

void crearBinario(Array caracter numeroBinario, num largo)
      FOR num i=0 TO largo-1
            numeroBinario[i] = '0'

void generarCombinaciones(Array Elemento listaElementos, num
cantElementos, Array caracter numeroBinario, num limite ponderacion, num
mayorSumaEncontrada)
      aumentarBinario(numeroBinario, cantElementos)
      num sumaValorActual = 0
      num sumaPonderacionActual = 0

      FOR num i=cantElementos-1 TO 0
            IF numeroBinario[i] = '1' THEN
```

```
                  sumaValorActual = sumaValorActual +
                  listaElementos[i].valor
                  sumaPonderacionActual = sumaPonderacionActual +
                  listaElementos[i].ponderacion
            IF sumaPonderacionActual > limitePonderacion THEN
                  RETURN

      IF sumaPonderacionActual <= limitePonderacion THEN
            IF sumaValorActual > mayorSumaEncontrada THEN
                  mayorSumaEncontrada = sumaValorActual

            RETURN

array Elemento leerArchivo(char nombreEntrada, direc num
cantidadElementos, direc num limitePonderacion, direct num
mayorSumaValoresEncontrada)
      FILE archivoEntrada
      num valor
      num ponderacion

      //Extraer la cantidad de elementos y límite de ponderacion del
      nombreEntrada
      cantidadElementos = leerArchivoEntrada(archivoEntrada)
      limitePonderacion = leerArchivoEntrada(archivoEntrada)
      array Elemento listaElementos[cantidadElementos]

      FOR i=0 TO cantElementos-1
            valor = leerElemento(archivoEntrada)
            ponderacion = leerElemento(archivoEntrada)
            listaElementos[i] = crearElemento(valor, ponderacion)

            IF valor > mayorSumaValoresEncontrada THEN
                  mayorSumaValoresEncontrada = valor

            i = i + 1

      RETURN listaElementos

MAIN:

      num cantidadElementos
      num limitePonderacion
      num mayorSumaEncontrada = 0
      Elemento listaElementos = leerArchivoEntrada(nombreArchivo,
cantElementos, limitePonderacion, mayorSumaEncontrada)

      Array caracter binarioCombinaciones[cantElementos]
      crearBinario(binarioCombinaciones, cantElementos)

      FOR num i=1 TO 2^cantElementos
```

```
            generarCombinaciones(listaElementos, cantidadElementos,
binarioCombinaciones, limitePonderacion, mayorSumaEncontrada)

      MOSTRAR mayorSumaEncontrada
```

Análisis de complejidad solución 2:

- crearElemento:

  $T(n) = 4$

  $O(1)$

- aumentarBinario:

  $T(n) = 3 + 4(n - 1)$

  $O(n)$

- crearBinario:

  $T(n) = 2n$

  $O(n)$

- generarCombinaciones:

  $T(n) = 6 + 3n$

  $O(n)$

- leerArchivo:

  $T(n) = 12 + 11n$

  $O(n)$

- Programa completo:

  $T(n) = 17 + 13n + 2^n + 2^n * (6 + 3n)$

  $O(2^n)$

6. REFERENCIAS

Pasto, M. (2008). El triángulo de Pascal o Tartaglia. (Recuperado el 12/05/2021) https://www.estadisticaparatodos.es/taller/triangulo/triangulo.html?f=15

Rodríguez, E. (2018). Algoritmos de fuerza bruta. (Recuperado el 12/05/2021) https://www.tamps.cinvestav.mx/~ertello/algorithms/sesion06.pdf