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

Estructura de datos y análisis de algoritmos

Laboratorio 1 - Salvación espacial

Christofer Rodríguez

Profesor: Mario Inostroza

Ayudante: Estaban Silva

Santiago – Chile
2-2020

Tabla de contenidos

Índice de figuras

Índice de tablas

CAPÍTULO 1. INTRODUCCIÓN

Debido a una pandemia mundial la doctora Ximi dueña de una nave espacial, ha tomado la decisión de migrar a otro planeta, debido a que esta nave posee una capacidad máxima, se debe tomar la decisión de que postulantes se convertirán en tripulantes en base a su calificación y peso. Para esto se debe construir un programa en el lenguaje de programación C que, a partir de un archivo con los datos de los postulantes y la nave, genere un archivo de salida con todas las posibles combinaciones de tripulantes que cumplan con el límite de peso, tanto las soluciones como los tripulantes serán mostrados de manera descendiente con respecto a su calificación.

Los objetivos de este informe son explicar la implementación de dicho programa, analizar la complejidad tanto del programa en general como de las funciones que lo componen, para finalmente se analizar el alcance y limitaciones de la solución empleada.

La manera de llevar a cabo estos objetivos será, primeramente, explicar los conceptos fundamentales para comprender este documento en el marco teórico, luego se explicará y analizará paso a paso tanto las estructuras, como los algoritmos utilizados para la implementación del programa en la descripción de la solución, además se estudiará su complejidad algorítmica, lo que nos permitirá llegar a expresiones matemáticas que ayudarán a entender su comportamiento. Para finalizar, en el análisis de resultados se examinarán las falencias, limitaciones, ventajas, la eficiencia algorítmica y se comparara el tiempo teórico con el tiempo real de ejecución del programa creado.

CAPITULO 2. DESCRIPCIÓN DE LA SOLUCIÓN

2.1.1 Marco teórico

- **Algoritmo:** Procedimiento que mediante una serie de pasos transforma una entrada o un conjunto de estas en una salida.

- **Búsqueda de fuerza bruta:** Algoritmo que consiste en generar todas las posibles soluciones a un problema, de esta manera se puede comprobar de entre todas las soluciones, cuales satisfacen las condiciones del problema.

- **Combinaciones sin repetición:** Las combinaciones consiste en estudiar las distintas formas de seleccionar un subconjunto de elementos pertenecientes a un conjunto, en el caso de las combinaciones sin repetición, no importa el orden de elección, esto quiere decir que no pueden existir dos subconjuntos con los mismos componentes.

- **Recursión:** Proceso que se define sobre su propia definición, específicamente en computación, hace referencia a que la solución de un algoritmo depende de instancias más pequeñas de este mismo.

- **Eficiencia algorítmica:** Análisis de la cantidad de recursos utilizados por un determinado algoritmo, estos recursos pueden ser, por ejemplo: tiempo o memoria.

- **Ordenamiento de burbuja:** Algoritmo de ordenamiento sencillo que compara un elemento de una lista con el siguiente, si estos se encuentran en el orden incorrecto, los intercambia. Esta revisión se realiza hasta que la lista este totalmente ordenada.

- **Struct:** Declaración de estructura en la cual se especifica una secuencia de variables que pueden ser de distinto tipo, posteriormente sirve como plantilla para la creación de esta estructura.

- **Arreglo:** Variables estructuradas, donde cada elemento se almacena de forma consecutiva dentro de la memoria, estos pueden ser de diferentes tipos de datos.

2.1.2 Algoritmos y estructura de datos

Para la solución de este problema se decidió utilizar un algoritmo de búsqueda de fuerza bruta, de esta manera se generarán todas las posibles soluciones y solamente aquellas que cumplan el límite de peso máximo que tiene la nave, serán consideradas como soluciones finales.

A continuación, se enumerará de manera las etapas que sigue el programa para encontrar la solución del problema, cada una de estas serán explicada de manera más detallada a lo largo de esta sección:

1. Leer la información contenida en un archivo de entrada.
2. Extraer la información y almacenarla en su respectiva estructura representativa.
3. Generar todas las posibles combinaciones de postulantes sin repetición.
4. Se comprueba que la combinación generada cumpla los requisitos para ser considerada una solución final.
5. Almacenar las soluciones en una lista.
6. Ordenar las soluciones de manera decreciente con respecto a su calificación
7. Generar un archivo de salida con las combinaciones ordenadas.

Tanto para la representación de los postulantes como las soluciones, se utilizó el tipo de dato **struct**, esta representación fue definida por el nombre **Postulante** y está compuesta por tres elementos:

El primero es estos, es un arreglo del tipo **char**, esta variable se utiliza para almacenar el nombre del postulante o la combinación del nombre de todos los tripulantes en caso de ser una solución, el segundo es un **int** en el cual se guardará el peso de un postulante o la suma de todos los integrantes de una solución, finalmente el tercero de ellos, es un **int** que contiene la calificación de un postulante o la suma total de quienes componen una combinación.

```
5   //Estructura que tendra cada postulante del archivo de entrada
6   typedef struct Postulante
7   {
8       char nombre[200];
9       int peso;
10      int calificacion;
11
12  } Postulante;
```

Ilustración 1: Estructura del tipo de dato Postulante

Para comenzar en la ejecución del programa, se crear 2 variables de tipo **int** que servirán para almacenar la cantidad de postulantes que hay en el archivo de entrada y el peso máximo de la nave, luego de crear estas variables, se procede a leer el archivo de entrada mediante la función **leer.**

```
Postulante* leer (char const* nombre, int *postulantes, int *pesoMax)
{
    FILE *pf;
    //Se abre el archivo en modo lectura
    pf = fopen(nombre, "r");
```

Ilustración 2: Función leer

Esta función se encarga de abrir el archivo de entrada en modo lectura y almacenar los datos de la nave, junto con los postulantes. Recibe como argumentos el nombre del archivo, el cual debe ser un arreglo de **char**, además se debe entregar la dirección de memoria de las variables donde se almacenará la cantidad de postulantes existentes y el peso máximo que soporta la nave. El retorno de esta función será un arreglo de **Postulante** con el contenido del archivo leído. En caso de que el archivo no exista, se generará un **Postulante** con un valor característico para identificar el error.

```
//En caso de que el archivo de entrada no exista
if (pf == NULL)
{
    Postulante * listaPostulantes = (Postulante*)malloc(sizeof(Postulante)*1);
    listaPostulantes[0].peso = -1;
    return listaPostulantes;
}
```

Ilustración 3: Elemento generado en caso de que el archivo no exista

En caso de que el archivo exista, se reservará memoria para un arreglo del tipo **Postulante** que contendrá a todos los postulantes del archivo de entrada, además se leerán y almacenarán los dos primero elementos del archivo que corresponden a la cantidad de postulantes, junto con la capacidad máxima de la nave; en caso de que estos datos sean inválidos (menores o iguales a 0), se retornará un **Postulante** con cierto valor en uno de sus componentes para identificar el error.

Después de comprobar que no existan errores en el archivo de entrada, se leerá el resto de este mediante un ciclo **while** mientras no se detecte que se llegó al final del archivo, dentro de este ciclo se creara un **Postulante** y se asignara cada valor en su respectiva variable; antes de agregarlos a la lista de postulantes, se comprueba que tanto el peso como la calificación sean valores valido (enteros positivos), en el caso de que no lo sean, al igual que los errores anteriores, se retornara cierto arreglo para identificar que hubo un problema.

```
//Ciclo para recorrer el archivo hasta llegar al final de este
while(feof(pf) == 0)
{
    //Se crea un nuevo postulante
    Postulante actual;

    //Se leen y almacenan los datos del postulante
    fscanf(pf, "%s", &actual.nombre);
    fscanf(pf, "%d", &actual.peso);
    fscanf(pf, "%d", &actual.calificacion);

    //En caso de que se ingrese un valor invalido en
    if (actual.peso <= 0 || actual.calificacion <= 0)
    {
        Postulante * listaPostulantes = (Postulante*)malloc(sizeof(Postulante)*1);
        listaPostulantes[0].peso = -2;
        return listaPostulantes;
    }
    //Se agrega el postulante a la lista de postulantes
    listaPostulantes[i] = actual;
    i += 1;
}
```

Ilustración 4: Asignación y comprobación de valores

Si no se encuentra ningún elemento invalido en los componentes de la estructura, este es agregado dentro del arreglo de postulantes creado previamente; para finalizar se cierra el archivo de texto y se retorna la lista de postulantes.

Este algoritmo **leer** posee un $T(n) = 11C + 6C * n$, por lo que se trata de un algoritmo con un orden lineal $O(n)$.

Terminado de leer el archivo de entrada y generado el arreglo con los postulantes, se procede a comprobar que no haya existido un error durante la lectura del archivo, en caso de que haya existido, el programa terminara de ejecutarse indicando la razón de la detención. Si es que no se encontró ningún fallo, se reserva memoria para un arreglo del tipo **int,** con un largo igual a la cantidad de postulantes presentes en el archivo de entrada, esto servirá para contener las posiciones que tienen los tripulantes dentro de la lista de postulantes que compondrán la solución, además se crea un arreglo de tipo **Postulante** con memoria inicial cero para almacenar las soluciones finales, también se inicializa una variable **int** que llevará la cuenta de la cantidad de soluciones presentes dentro del arreglo.

La siguiente sección del programa es la más importante, ya que es la encargada de realizar las distintas combinaciones para el problema. De manera exterior se utilizan dos ciclos **for**, uno dentro de otro; el exterior sirve para llevar la variar la cantidad elementos que debe contener la solución actual, mientras que el interior sirve para seleccionar el “pivote” de la combinación. Una vez dentro de los ciclos, se asigna el valor del pivote en la primera posición de la lista de posiciones y se comienza a generar todas las posibles combinaciones mediante el algoritmo llamado **combinación**.

```
//Ciclo que lleva la cuenta de cuantos elementos debe tener la combinacion del ciclo correspondiente
for (int cantElementos = 1; cantElementos <= postulantes; cantElementos++)
{

    //Ciclo para recorrer los elementos de una lista de postulantes, la variable i sirve para elegir el pivote de la combinatoria
    for (int i = 0; i < postulantes; i++)
    {

        //Se almacena el valor del "pivote" como primer elemento del arreglo de posiciones
        posiciones[0] = i;

        //Se llama la funcion para generar las combinaciones de postulantes
        soluciones = combinacion(posiciones, i+1, 1, cantElementos, nave, postulantes, nave[i].peso, nave[i].calificacion, pesoMax, soluciones, &cantSoluciones);
    }
}
```

Ilustración 5: Llamada del algoritmo combinación

La función **combinación**, es un algoritmo recursivo que se encarga de generar todas las combinaciones sin repetición con los índices de las posiciones, estos van desde el 0 hasta $n-1$, recibe como argumentos la lista de posiciones, la siguiente posición a agregar, la longitud de la combinación actual (inicialmente 1), la cantidad de elementos que debe tener la posible solución a generar, la lista de postulantes, la cantidad de elementos que tiene la lista de postulantes, el peso del pivote, la calificación de este, el peso máximo de la nave, el arreglo donde se guardan las soluciones finales y la dirección de memoria de la variable que lleva la cuenta del largo de la lista de soluciones.

Como algoritmo recursivo, este cuenta con un caso base y un caso general. El caso base se da cuando la cantidad de elementos de la combinación actual es igual al largo que debe tener esta solución, antes de considerarla una solución final, se comprueba que el peso de esta combinación no supere el límite de la nave, si lo sobrepasa se descartará, pero si cumple con la capacidad de la nave, se creara un **Postulante** para representar la solución con la ayuda del algoritmo **crearSolucion**.

Dicha función se encarga de crear una variable del tipo **Postulante** que contenga a los postulantes en las posiciones almacenadas en la lista de posiciones, recibe como argumentos la lista de posiciones, el largo que tiene la combinación actual, la lista de postulantes, la suma del peso y la suma de la calificación de los tripulantes que compondrán esta solución. Primero se crea un arreglo de **char** para con espacio para 260 caracteres que usaremos para contener la unión del nombre de los tripulantes, después mediante un ciclo **for** recorremos cada posición de la lista de posiciones, valor que utilizaremos como índice para agregar el nombre del postulante dentro de la lista. Una vez terminado de unir los nombres, se crea un **Postulante** la combinación de todos los componentes de la solución empleando la función **crearPostulante**, función constructora de un **Postulante**; finalmente esta tripulación en añadida a la lista de soluciones y se actualiza la cantidad de elementos que esta contiene.

La función **crearSolucion** posee un $T(n) = 5C + 2C * n + 2n^2$, y un $O(n^2)$.

```
Postulante crearSolucion (int* posiciones, int largo, Postulante* postulantes, int peso, int calificacion)
{
    //Se crea una variable (char) auxiliar que contendra el nombre final de la solucion (union de todos los postulantes)
    char auxiliar[260];
    //Se iguala la variable variable auxiliar a un string vacio para que no se altere el nombre
    strcpy(auxiliar,"");

    //Ciclo que recorre los elementos del arreglo posiciones
    for(int b = 0; b < largo; b++)
    {
        //Une el nombre acumulado en la variable auxiliar con el nombre del postulante en la posicion actual
        strcat(auxiliar, postulantes[posiciones[b]].nombre);

        //Si es que queda otro elemento a agregar se agregara una coma al final del stirng para separar el nombre de los postulantes
        if (b+1 < largo)
        {
            strcat(auxiliar, ", ");
        }
    }

    Postulante solucion = cearPostulante(auxiliar, peso, calificacion);
    free(auxiliar);

    return solucion;
}
```

Ilustración 6: creación de una solución en el caso base

Con la solución ya creada, se llama a la función **agregarSolucion,** este algoritmo está encargado de recibir una lista de soluciones, la cantidad de elementos que posee, la combinación que se desea agregar, y crear un nuevo arreglo de tipo **Postulante** actualizado.

Para llevar a cabo esta tarea, primero reserva memoria para una lista de **Postulante**, del largo actual más un elemento, luego mediante un ciclo **for** se copia cada combinación de la lista original al nuevo arreglo. Una vez copiada, se procede a almacenar los datos de la nueva solución en la última posición, se actualiza el valor de la variable que contiene el largo, se libera la memoria ocupada por el antiguo arreglo y retorna el nuevo.

Ya que el algoritmo **agregarSolucion**, en el peor de sus casos tendrá que copiar todas las combinaciones posibles, tiene un $T(n) = 7C + 2^n * n + 3C * n$, con un $O(2^n)$.

```
Postulante* agregarSolucion (Postulante * listaSoluciones, int *largoLista, Postulante solucionAgregar)
{
    Postulante * nuevaLista = (Postulante*)malloc(sizeof(Postulante)*((*largoLista)+1));  //Se crea una nueva lista con el tamanio nuevo

    //Copiamos los elementos de la lista antigua a la nueva
    for (int i = 0; i < *largoLista; i++)  //Ciclo para recorrer los elementos de la lista
    {
        strcpy(nuevaLista[i].nombre, listaSoluciones[i].nombre);
        nuevaLista[i].peso = listaSoluciones[i].peso;
        nuevaLista[i].calificacion = listaSoluciones[i].calificacion;
    }

    //Ahora copiamos la nueva solucion al final de la lista
    strcpy(nuevaLista[*largoLista].nombre, solucionAgregar.nombre);
    nuevaLista[*largoLista].peso = solucionAgregar.peso;
    nuevaLista[*largoLista].calificacion = solucionAgregar.calificacion;

    free(listaSoluciones);  //Liberamos la memoria utilizada por la antigua lista
    *largoLista = *largoLista + 1;  //Actualizamos el largo de la lista

    return nuevaLista;  //Retornamos la nueva lista
}
```

Ilustración 7: Estructura del algoritmo agregarSolución

Por otro lado tenemos el caso general del algoritmo recursivo, situación donde la combinación actual no tiene la cantidad de elemento que debería tener, mediante un ciclo **for** se hace variar el valor del siguiente índice y luego de agregar la nueva posición a la lista, luego se hace el llamado recursivo, modificando el valor de la siguiente posición incrementada en 1, se actualiza la longitud actual de la combinación, se suma el peso del nuevo postulante agregado, se incorpora a la calificación anterior la del nuevo postulante y el resto de los parámetro se mantienen constantes. De esta manera se agregan elementos hasta llegar a la longitud deseada, una vez agregada esta combinación, se vuelve al ciclo **for** del caso general y se hace variar el último elemento hasta haber tomado todos los componentes del conjunto, cuando no quedan más posiciones, se modifica el pivote y se combina con el resto.

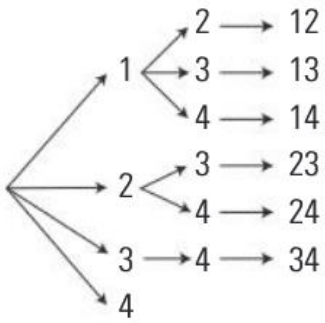

Ilustración 7: Árbol de combinación sin repetición $C_{4,2}$

Habiendo generado todas las combinaciones con cierta cantidad de elementos, el ciclo **for** externo se encarga de hacer variar este valor, de esta manera se logra generar todas las posibles soluciones al problema.

El algoritmo de combinación cuenta con un $T(n) = (9C * n + 2C * n^2 + 2n^3)\ 2\ (2^n - 1)$, y $O(2^n)$ para el peor de sus casos.

Luego de obtener todas las soluciones que cumplen con las condiciones del problema, llamamos al algoritmo de ordenamiento de burbuja para organizar las combinaciones desde la que tenga mayor calificación, hasta la que posea menos. Este algoritmo consiste en dos partes, primero mediante el uso de dos ciclos **for** anidados, se compara el valor de una posición del arreglo con la siguiente y si estos se encuentran en el orden equivocado (en este caso con respecto a la calificación), estos intercambian su posición; el encargado de realizar este cambio es la función **swap**.

Esta función recibe como argumentos el arreglo que se desea ordenar y la cantidad de elementos que posee, una vez ordenado, no retorna nada ya que modifica directamente dicho arreglo.

El algoritmo de ordenamiento de burbuja tiene un $T(n) = 2C + 4C * n^2$, y un $O(n^2)$.

Teniendo todas las combinaciones generadas y ordenadas con la función **generarSalida** se crea un archivo **FILE** en modo escritura, de esta manera si el archivo existe previamente, este será remplazado y si este no es el caso, simplemente lo creará. Esta función recibe como argumentos el arreglo donde están asignadas las combinaciones y la cantidad de elementos que este arreglo contiene.

Mediante ciclo un **for**, recorremos la lista de soluciones y escribimos en cada línea del archivo de salida el nombre de los tripulantes, la suma total de peso y la calificación total de cada solución, ya que previamente ordenamos esta lista, las soluciones serán escritas desde la que tenga mayor calificación, hasta la que posea menos. Antes de terminar la ejecución de la función, se cierra el archivo de salida y este algoritmo no retorna nada, solamente genera el documento de texto plano. Para finalizar la ejecución del programa, se libera la memoria utilizada por los principales arreglos.

La función de **generarSalida** en el peor de los casos, dispone de un $T(n) = 3C + (2^n - 1) * 4C * n$, con un orden exponencial $O(2^n)$.

```
void generarSalida (Postulante* listaSoluciones, int cantSoluciones)
{
    FILE* salida;
    //Se abre el archivo "tripulacion.out" en modo escritura (lo creara si no existe o lo remplazara si existe)
    salida = fopen ("tripulacion.out", "w");

    //Ciclo para escribir cada solucion de la lista de soluciones en una linea
    for(int i = 0; i < cantSoluciones; i++)
    {

        //Se escribe la solucion
        fprintf(salida, "%s, %d, %d\n", listaSoluciones[i].nombre, listaSoluciones[i].peso, listaSoluciones[i].calificacion);
    }

    //Se cierra el archivo de salida
    fclose(salida);

    return;
}
```

Ilustración 8: Estructura de la función generarSalida

2.2 Análisis de resultados

El programa creado resuelve el problema de manera completa, este es capaz de leer un archivo de entrada, generar todas las combinaciones posibles de postulantes sin repetición, filtrar las soluciones que cumplen los requisitos del problema, ordenar tanto los postulantes como las combinaciones de manera decreciente con respecto a su calificación en un archivo de salida.

A pesar todo lo anteriormente nombrado, debido a la manera de abordar el problema, el programa también tiene sus falencias.

2.2.1 Manejo de errores

El principal momento donde se pueden encontrar errores, es al momento de leer el archivo de entrada, debida a que el algoritmo encargado de leer dicho archivo debe retornar un tipo un arreglo de **Postulante**, los problemas que pueden surgir no son manejados de manera correcta, ya que se debe crear un arreglo y de alguna manera establecer algo que haga notar que ocurrió dentro del algoritmo, esto limita el manejo de errores a una forma ineficiente de hacerlo.

2.2.2 Manejo de memoria

Si bien la búsqueda por fuerza bruta en una manera eficaz y simple de abordar un problema no siempre es la mejor en cuanto al uso de memoria, ya que se deben generar todas las posibles soluciones del problema, que en este caso en específico no es un número reducido. Otro pequeño problema en cuanto al manejo de memoria es el trabajar con arreglos estáticos para el nombre de los postulantes, ya que C no tiene un buen manejo de **strings**, se debe utilizar arreglos del tipo **char** con una capacidad previamente definida, por esto cuando trabajemos con nombres pequeños, estaremos desperdiciando memoria y cuando tengamos combinaciones de **strings** muy largos, nos faltará memoria.

2.2.3 Complejidad algorítmica

Luego de calcular la complejidad algorítmica da cada algoritmo utilizado en la ejecución del programa, se llego a la siguiente expresión matemática:

$$T(n) = 29C + 6Cn + 12Cn^2 + (9Cn^3 + 2Cn^4 + 2n^5)\,2(2^n - 1) + (2^n - 1)$$

Esta expresión representa la cantidad de instrucciones que tendrá que ejecutar el programa para poder obtener todas las soluciones en el peor de los casos.

De lo anterior también podemos obtener que el orden del programa es $O(2^n)$, a pesar de que se dejaron de lado expresiones de alto orden, este es el orden que caracteriza el algoritmo usado.

Como podemos ver en las siguientes ilustraciones, un algoritmo de orden 2^n, crece muchísimo con una pequeña variación en la cantidad de elementos de entrada, tanto en niveles bajos como altos.

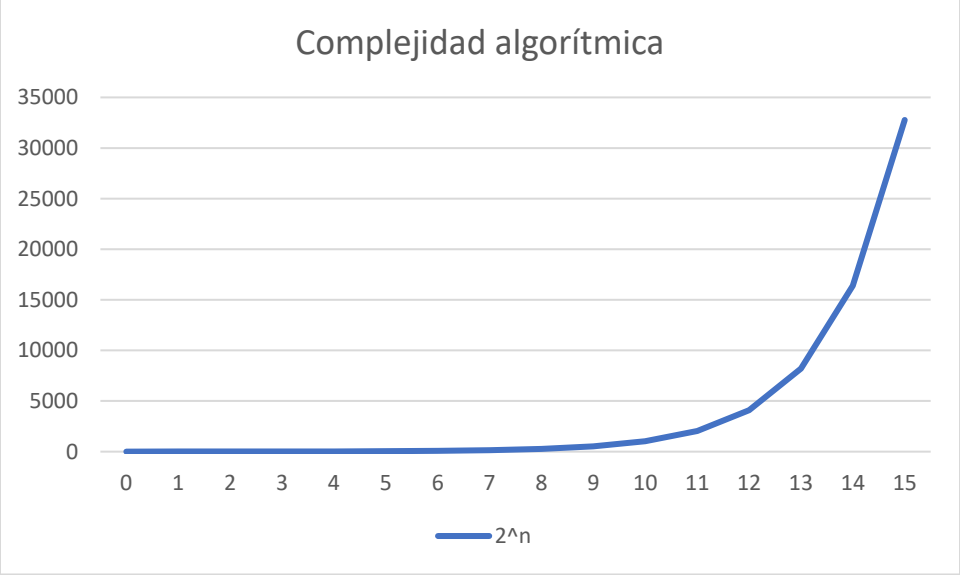

Ilustración 9: Grafico de la complejidad algorítmica con entradas pequeñas

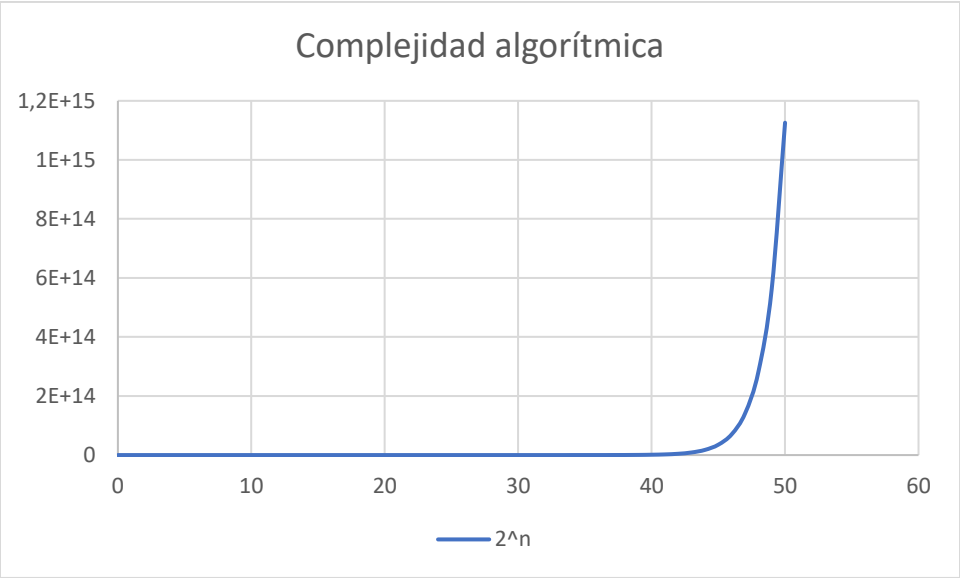

Ilustración 10: Grafico de complejidad algorítmica con entradas grandes

Para una mayor comprensión del tiempo de ejecución del programa, se registro el periodo de ejecución con distintos tamaños de entrada. Esto se hizo agregando los siguientes elementos dentro del código:

Primero se importó la librería **time.h**, se crearon las variables para almacenar la hora al momento del inicio y al momento del término. Esta primera variable es inicializada con la hora del equipo antes de comenzar a leer el archivo de entrada.

```
clock_t tiempo_inicio, tiempo_final;
double segundos;
tiempo_inicio = clock();
```

Ilustración 11: Variables para tomar el tiempo

Cuando termina la generarse el archivo de salida se almacena la hora del equipo en la variable **tiempo_final** y se muestra por pantalla la diferencia entre las horas.

```
tiempo_final = clock();
segundos = (double)(tiempo_final - tiempo_inicio) / CLOCKS_PER_SEC;
printf("%f",segundos);
```

Ilustración 12: Obtención del tiempo de ejecución

El tiempo obtenido lo podemos ver en la siguiente tabla:

Tabla 1: Tamaño de entrada vs tiempo de ejecución

Tamaño de entrada	Tiempo real (segundos)
5	< 1
10	0.007
15	0.165
20	8.064
25	312.19

Al igual que en el gráfico, podemos ver claramente que debido al orden de complejidad que posee el programa, el haber elegido generar todas las combinaciones posibles mediante la búsqueda por fuerza bruta, no fue la mejor solución; a pesar de haber añadido un algoritmo que copiaba la lista de soluciones en una nueva al momento de agregar una solución, esto termino mejorando el uso de memoria, pero a la vez aumentando considerablemente la cantidad de instrucciones que debe ejecutar el programa.

2.3 Conclusión

Si bien se logró el resolver el problema planteado de manera completa, mediante el análisis de complejidad algorítmica al programa, se puede notar el elevado tiempo de ejecución y su deficiencia en el uso de memoria, a pesar de todo esto, este laboratorio fue muy importante, ya que, al explicar y analizar cada parte de la manera de abordar el problema, se puede observar las ventajas y desventajas de cada algoritmo utilizado, lo cual será algo a considerar en futuros laboratorios o instancias en las que se tenga que programar. Como principal resultado se puede decir que es de vital importancia considerar cada aspecto del problema para llegar a una solución que combine eficiencia en el uso de memoria y en el tiempo de ejecución, porque no siempre la solución más simple o clara es la más adecuada.

Referencias

Arias, J. Maza, I. (2012). Solucionario Matemáticas Bruño 4-ESO-B. España: Grupo Editorial Bruño, S. L.

EcuRed (2012). Ordenamiento de burbuja. (Recuperado 16/11/2020) https://www.ecured.cu/Ordenamiento_de_burbuja

Microsoft (2016). Declaraciones de estructura. (Recuperado 16/11/2020) https://docs.microsoft.com/es-es/cpp/c-language/structure-declarations?view=msvc-160

Pasto, M. (2008). El triángulo de Pascal o Tartaglia. (Recuperado 16/11/2020) https://www.estadisticaparatodos.es/taller/triangulo/triangulo.html?f=15

REAL ACADEMIA ESPAÑOLA: *Diccionario de la lengua española*, 23.ª ed., [versión 23.3 en línea]. (Recuperado 15/11/2020). https://dle.rae.es