Universidad Del Valle

Escuela de Ingeniería de Sistemas y Ciencias de la Computación

El problema del Zoológico de Cali

Profesor:
Jesús Alexander Aranda

Estudiantes:
Nicolás Lasso (1740395), Jorge Mayor(1738661), Cesar Becerra(1744338)

Mayo 2020

Introducción

En este documento se realiza una explicación de las soluciones dadas al problema propuesto como proyecto final de la clase de Fundamentos y análisis de diseño de algoritmos, en el que se plantea ordenar un hipotético show, del zoológico de la ciudad de cali, teniendo como punto de comparación la grandeza que tiene cada animal que es partícipe del espectáculo. También se suministran ciertas variables a tener en cuenta, como los son la cantidad de animales (n), la cantidad de partes del show (m) y la cantidad de escenas de cada parte del show (k) diferente de la apertura de la misma, la cual tiene más partes ((m-1)*k). Se pide realizar 3 soluciones con distintas complejidades (lineal, nlog(n) y cuadrática). Primero se habla del método de entrada utilizado y se procede a explicar las variables y funciones utilizadas en las 3 soluciones, haciendo después una aclaración de las diferencias que hay entre una y otra. Finalmente se hace una análisis de resultados, en el que se comparará el rendimiento de las distintas soluciones. .

1. Implementación de las soluciones

1.1. Entradas

Como entrada del problema se utilizan archivos.txt con un formato determinado que se explicará a continuación. La primera línea consiste en tres enteros, $1 \leq N \leq 3*M*K$, el número de animales; $1 \leq M \leq 60$, el número de espectáculos; y $1 \leq K \leq N$, el número de escenas por espectáculo, seguido de n líneas con una cadena de caracteres S (incluye mayúsculas y minúsculas) separada por un espacio de un entero $1 \leq G \leq N$, equivalente a su grandeza. Está garantizado que la grandeza de cada animal es diferente.

Posteriormente se tiene un párrafo, con (m-1)k líneas, correspondiente a la apertura, en cada una hay 3 cadenas de caracteres iguales a algunas de las n cadenas citadas anteriormente, cada una de estas líneas corresponde a una escena de la apertura. Cabe aclarar que todas las n cadenas (animales) deben ser usadas en al menos una escena de la apertura.

Luego se tienen (m-1) párrafos con k líneas, que corresponden a las partes posteriores a la apertura. Las líneas de estas partes son extraídas de las líneas de la apertura, todas las líneas de la apertura deben ser usadas como mínimo una vez en estos párrafos.

1.2. Variables Globales

Ahora, se hablará de las distintas variables usadas en las soluciones planteadas.

n: la variable n representa la cantidad de animales que van a hacer parte del show. n es un entero que puede adquirir un valor máximo de 3*m*k y mínimo de 1.

m: La variable m representa las partes de las que se compone el show. m es un entero que puede adquirir un valor máximo de 60 y mínimo de 1.

k: La variable K representa la cantidad de escenas que van a tener las parte del show, excluyendo la apertura, la cual tiene (m-1)*k partes. k es un entero que puede adquirir un valor máximo de n y mínimo de 1.

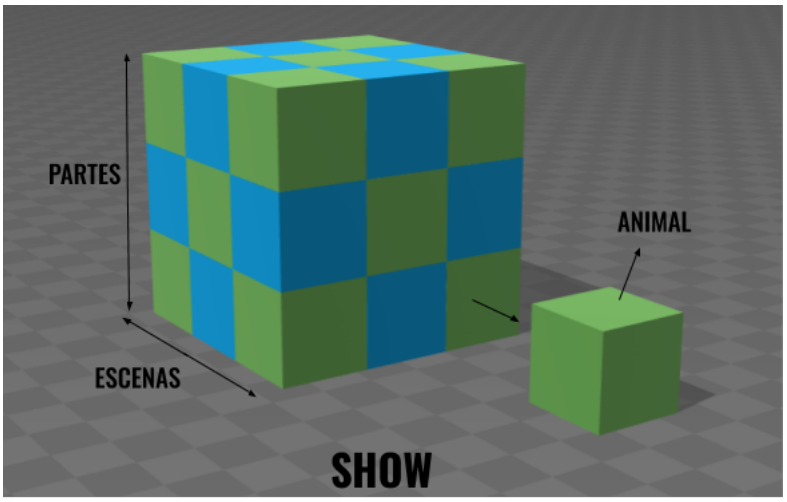
animals: Esta variable es un unordered_map (mapa sin orden) en el que se almacenan cada animal con su respectiva grandeza. El unordered_map funciona como un diccionario en c++. Hash table es una función utilizada para implementar los diccionarios en c++, en ella cada cadena de caracteres que le es suministrada se le realiza un proceso, el cual arroja como resultado un número que apunta a un valor, este valor debe estar asociado a la grandeza del animal representado por esa cadena de caracteres. Cada uno de los animales es mapeado a su respectiva grandeza por medio de esta función, la complejidad necesaria para acceder a cada una de esas grandezas es O(1) gracias al Hash table.

Es posible que la función Hash table tenga una complejidad mayor, pero esto solo ocurre si se presentan colisiones. Una colisión ocurre si dos números generados por distintas cadenas de caracteres apuntan a la misma dirección, en este caso no solo se puede acceder al dato en la dirección, si no que se tiene que ver cual de los dos datos contenidos en esa dirección es el correspondiente. En el peor de los casos, que hayan muchas colisiones, el comportamiento terminaría siendo O(n).

Se asumió que no se van a generar colisiones, ya que es muy poco probable, y de esta forma solo se tendrá una complejidad O(1).

animalParticipations: Es una estructura igual a la anterior, pero esta vez es utilizada para almacenar la cantidad de participaciones de cada animal. Al ser la misma estructura que la variable animals se presenta la misma complejidad O(1) con el mismo limitante de que si se presentan n colisiones puede llegar a ser O(n). Prácticamente, para que se presente una colisión dos datos deben ser manipulados con tal de que se mapeen en el mismo valor.

show: Esta estructura es un vector de vector de vectores de strings, se puede ver como una representación de un cubo, por un lado se tienen 3 animales correspondientes a una escena, por otro lado tenemos las escenas que conforman una de las partes, y por el otro lado tenemos las escenas que conforman el show.



showAwesomeness: Este es un entero que indica la grandeza general de todo el show, es la suma de las grandezas de todas las escenas, y es calculado dentro de la función encargada de ordenar las partes.

1.3. Funciones

setAnimals: Esta función es la encargada de leer un archivo, el cual se le es suministrado como entrada, que contiene los animales de un caso de pruebas específico del directorio “Pruebas”.

Como ya se mencionó, la función setAnimals recibe como parámetro un archivo del directorio de pruebas. La librería fstream permite la lectura y asignación de los datos contenidos en el texto a distintas variables creadas en la función, obteniendo así una representación de los animales contenidos en el archivo del caso de pruebas específico.

```
void setAnimals(ifstream & file) {

        string name;
        int awesomeness;
        for(int i=0; i<n; i++){
                file >> name >> awesomeness;
                animals[name] = awesomeness;
        }
}
```

Se evidencia que la complejidad de esta función se debe al bucle For. Como este último hace un recorrido de 0 hasta n, se puede decir que tiene una complejidad lineal, que además está acompañada por una constante (2) dada por las operaciones que se realizan dentro del mismo:

$$T(n) = 2n$$

Ya que se tiene como objetivo predecir el comportamiento de la función en el infinito podemos prescindir de la constante, esto ocurre porque al evaluar la función con datos muy grandes los valores obtenidos al contar o no con la constante empiezan a ser similares debido a la clara dominancia de n sobre la constante. Según lo anterior se concluye que la función setAnimals tiene una complejidad lineal:

$$O(n)$$

setShow: Se encarga de leer el archivo de prueba y almacenar en un arreglo las distintas partes del show dado por el caso de prueba específico que se está trabajando.

La función setShow al igual que setAnimals recibe como parámetro el archivo del directorio de pruebas, y por medio de fstream los datos de dicho archivo son almacenados en el arreglo.

```
void setShow(ifstream & file){

        vector<vector<string>> part;
        vector<string> scene;
        string animal;

        for(int i=0; i<k*(m-1); i++){
```

```
            scene.clear();

            file>>animal;
            scene.push_back(animal);
            animalParticipations[animal]++;
            file>>animal;
            scene.push_back(animal);
            animalParticipations[animal]++;
            file>>animal;
            scene.push_back(animal);
            animalParticipations[animal]++;

            part.push_back(scene);
    }

    show.push_back(part);

    for(int i=1; i<m; i++){
            part.clear();

            for(int j=0; j<k; j++){
                    scene.clear();

                    file>>animal;
                    scene.push_back(animal);
                    animalParticipations[animal]++;
                    file>>animal;
                    scene.push_back(animal);
                    animalParticipations[animal]++;
                    file>>animal;
                    scene.push_back(animal);
                    animalParticipations[animal]++;

                    part.push_back(scene);
            }
            show.push_back(part);
    }
```

Esta función se divide en dos ciclos ya que se hace una distinción entre la lectura de la apertura del show y las partes posteriores a esta. Esto ocurre porque la apertura cuenta con más escenas que el resto de las partes del show.

A la hora de evaluar la complejidad se miran las dos partes por separado. El primer ciclo hace un recorrido desde 0 hasta k(m-1), y en el planteamiento del problema se dejó claro que m y k como máximo puede ser 60 y n, respectivamente, entonces se puede decir que la la función de complejidad de este ciclo es:

$$T_1(n) = n(c)$$

Se sabe que n es dominante sobre una constante, entonces se puede decir que este ciclo tiene una complejidad lineal:

$$O_1(n).$$

En el caso de la segunda parte de setShow se puede observar que está compuesta principalmente por dos ciclos for, uno contenido dentro del otro. Se puede pensar que por deberse a dos ciclos for anidados se está tratando con una complejidad cuadrática, pero hay que tener claro que el ciclo externo va de 0 a m y el interno de 0 a k. Como se dijo anteriormente m y k como máximo puede ser 60 y n, respectivamente. Por lo anterior dicho se observa que la complejidad en el peor de los casos es:

$$T_2(n) = (c)n$$

Y teniendo claro que n es dominante sobre una constante, la complejidad de la segunda parte de setShow es lineal:

$$O_2(n)$$

La función de complejidad de todo el programa se puede encontrar sumando la de cada parte:

$$T_t(N) = T_1(n) + T_2(n) = (c)n + (c)n = (2c)n$$

y como n es dominante, se dice que la complejidad total de toda la función es:

$$O(n)$$

getSceneAwesomeness: Dada una escena calcula la grandeza de los 3 animales que la componen. Esta función recibe como parámetro un vector que representa una escena y retorna un entero que es igual a la suma de la grandeza de los animales pertenecientes a dicha escena. Debido a que en la complejidad nlogn, los accesos a la estructura animals ahora son logarítmicos, la complejidad resultante de esta función es igual a T(n) = 3*log(n), que con n tendiendo al infinito es igual a O(log(n)).

```
void setAnimals(ifstream & file) {

int getSceneAwesomeness(vector<string> scene){

        int result = animals[scene[0]] + animals[scene[1]] + animals[scene[2]];

        return result;
}
```

Anteriormente se explicó que la complejidad que se supone para obtener un valor de un diccionario es O(1), entonces obtener las 3 valores sería:

$$T(n) = 3$$

Es decir, esta función es de complejidad constante:

$$O(1)$$

getPartAwesomeness: Dada una parte se calcula la grandeza de esta sumando la grandeza de cada escena que la compone. Esta función recibe como parámetro un vector de vectores de strings que representan una parte del show y retorna como resultado un entero que representa la grandeza total de la parte, la cual es la suma de la grandeza de cada escena que la compone. Debido al aumento de complejidad de la función getSceneAwesomeness en la solución de complejidad nlogn, la complejidad de esta función también aumenta, viendose incrementado log(n) veces, resultando con una complejidad con n tendiendo al finito de O(n*log(n)).

```
int getPartAwesomeness(vector<vector<string>> part){

    int result = 0;

    for(int i=0; i<part.size(); i++)
        result += getSceneAwesomeness(part[i]);

    return result;
}
```

La complejidad de esta función depende de si se le suministra como entrada la apertura o una de las partes que le siguen. Aunque en los dos casos la complejidad depende del bucle for, si se suministra la apertura como entrada el recorrido del for va de 0 a (m-1)*k, si es una de las partes siguientes sería igual a k. Como la operación ejecutada dentro del for es getSceneAwesomeness esta se multiplica con la complejidad obtenida por el for, generando:

Caso apertura: $T(m,k) = ((m-1)*k)3$

Caso parte: $T(k) = k*3$

y como se explicó anteriormente m y k como máximo puede ser 60 y n respectivamente, teniendo esto claro, la complejidad para los dos casos sería lineal:

$$O(n)$$

1.4. Ordenamiento en las soluciones

A continuación se describen los algoritmos de ordenamientos usados en las implementaciones y las funciones en las que se implementaron dichos algoritmos.

1.4.1 Solución O(n^2)

Para esta solución se implemento el algoritmo bubble sort como algoritmo de ordenamiento, que como se sabe, es de complejidad $O(n^2)$. Este se implemento en las funciones *sortScenes*, y la función *sortParts*. Los animales dentro de cada escena se ordenaron con la función *sortAnimals*.

sortAnimals: Esta función es encargada de organizar todos los animales dentro de cada escena del show. Está compuesta por dos ciclos for anidados; el exterior se encarga de extraer cada una de las partes del show, para extraer las escenas de dichas partes en el ciclo interior. Por cada una de las iteraciones del ciclo interior, se extraen las grandezas de los animales que hacen parte de la escena actual, gracias al diccionario creado en setAnimals, posteriormente comparándose en una serie con condicionales que se encargarán de organizar a los animales de la variable global show.

Debido a que el acceso al diccionario, por cada uno de los animales, es supuesto como constante, y que la cantidad máxima de comparaciones que pueden ser realizadas es igual a tres, la complejidad dentro del ciclo for interno es igual a O(1). Este mismo ciclo, parte de un iterador igual a cero y llega hasta de la parte que se está evaluando. Puesto que la apertura es la única parte con un tamaño diferente, la organización de sus animales será considerada aparte, teniendo una complejidad de T(m,k)=(k*(m-1)). Para las demás escenas, el iterador del ciclo exterior partiría desde 1, llegando hasta m, siendo m - 1 el total de iteraciones, mientras que el del ciclo interno, continúa variando entre cero y el tamaño de las partes, en este caso k, produciendo una complejidad de T(m,k)= (k*(m-1)) = (mk + k). Al final, la expresión que describiría la complejidad de toda la función vendría dada por T(m,k)= (2*k*(m-1)) = (2mk + 2k). Puesto que el término con mayor valor en la expresión es 2mk, se analizan los máximos valores que dicho términos pueden adoptar, siendo que m puede llegar a valer máximo 60 y k adopta como máximo el valor de n, la complejidad puede ser traducida como T(n) = 120n y, analizando n como un límite al infinito, O(n).

```
void sortAnimals() {

    vector<vector<string>> part;
    vector<string> scene;

    for(int i=0; i<m; i++) {

        part = show[i];
        for(int j=0; j<part.size(); j++) {

            scene = part[j];
```

```
            int awesomenessFirstAnimal = animals[scene[0]];
            int awesomenessSecondAnimal = animals[scene[1]];
            int awesomenessThirdAnimal = animals[scene[2]];

            if(awesomenessFirstAnimal < awesomenessSecondAnimal) {

                if(awesomenessSecondAnimal < awesomenessThirdAnimal) {

                    continue;

                } else if(awesomenessFirstAnimal < awesomenessThirdAnimal) {

                    swap(show[i][j][1], show[i][j][2]);
                } else {

                    swap(show[i][j][1], show[i][j][2]);
                    swap(show[i][j][0], show[i][j][1]);
                }
            } else if(awesomenessFirstAnimal < awesomenessThirdAnimal) {

                swap(show[i][j][0], show[i][j][1]);

            } else if(awesomenessSecondAnimal < awesomenessThirdAnimal) {

                swap(show[i][j][0], show[i][j][2]);
                swap(show[i][j][0], show[i][j][1]);

            } else {

                swap(show[i][j][0], show[i][j][2]);
            }
        }
    }
}
```

sortScenes: para esta complejidad $O(n^2)$ esta función únicamente ordena las escenas de una determinada parte con respecto a su grandeza total utilizando el algoritmo bubble sort.

```
void sortScenes() {

        for(int i=0; i<m; i++) {

                vector<vector<string>> part = show[i];
                vector<int> sortedIndexes[3*n-2];

                for(int j=0; j<part.size(); j++) {

                        vector<string> scene = part[j];
```

```
                int awesomeness = getSceneAwesomeness(scene);
                sortedIndexes[awesomeness].push_back(j);
        }

        vector<vector<string>> sortedPart;
        for(int j=6; j<=3*n-3; j++) {

                vector<int> scenesWithEqualAwesomeness = sortedIndexes[j];

                for(int l=0; l<scenesWithEqualAwesomeness.size(); l++) {
                        sortedPart.push_back(part[scenesWithEqualAwesomeness[l]]);
                }
        }
        show[i] = sortedPart;
    }
}
```

sortParts: esta función también utiliza el algoritmo bubble sort para ordenar todas las partes (incluida la apertura) del show.

```
void sortParts() {

    int partAwesomeness[m];
    int biggestPartAwesomeness = 0;

    for(int i=0; i<m; i++) {

        int awesomeness = getPartAwesomeness(show[i]);
        partAwesomeness[i] = awesomeness;
        showAwesomeness += awesomeness;

        if(awesomeness > biggestPartAwesomeness)
            biggestPartAwesomeness = awesomeness;
    }

    vector<int> sortedIndexes[biggestPartAwesomeness+1];

    for(int i=1; i<m; i++) {

        vector<vector<string>> part = show[i];
        int awesomeness = partAwesomeness[i];

        sortedIndexes[awesomeness].push_back(i);
    }

    vector<vector<vector<string>>> sortedShow;
    sortedShow.push_back(show[0]);
```

```
        for(int i=6; i<=biggestPartAwesomeness; i++) {

                vector<int> partsWithEqualAwesomeness = sortedIndexes[i];

                for(int j=0; j<partsWithEqualAwesomeness.size(); j++) {
                        sortedShow.push_back(show[partsWithEqualAwesomeness[j]]);
                }
        }

        show = sortedShow;
}
```

1.4.2 Solución O(nlog(n))

Para esta solución en particular, el tipo de datos de la variable global animals fue cambiado de unordered_map a map, haciendo que esta estructura ya no estuviera implementada sobre una tabla hash, sino sobre un árbol, provocando que los accesos de complejidad constante (en su mayoría) se convirtieran en accesos de complejidad logarítmica, lo que provoca un aumento en distintas funciones que se especificarán más adelante. De esta forma, las colisiones provocadas por la tabla hash son evitadas, obteniendo un complejidad que con certeza se puede acotar.

sortScenes: el propósito de esta función es organizar todas las escenas dentro de cada parte del show, utilizando para esto el algoritmo de ordenación counting sort.

Este algoritmo ordena una serie de datos, tomando en cuenta el valor máximo que estos pueden adoptar. En este caso específico, el algoritmo se encargará de almacenar cada uno de los índices de las escenas de una parte específica, en una posición de un arreglo, asociada al valor de su grandeza total.

Esta función está compuesta principalmente por un ciclo for, que dentro de sí tiene otros tres ciclos for, no anidados. El primero de estos de se encarga de encontrar el máximo valor de grandeza que pueden tomar las escenas, para así poder crear un arreglo que almacene las escenas ordenadas de una parte, sin exceder demasiado su tamaño. A su vez, se guarda el valor de la grandeza total de mayor magnitud de una escena, en la variable global biggestSceneAwesomeness. El iterador de este ciclo varía desde cero al tamaño de la parte analizada. Puesto que obtener la grandeza total de una escena es una operación logarítmica por la estructura de la variable animals, la complejidad de este ciclo para la apertura sería igual a $T(k, m, n) = (k * (m - 1) * log(n))$ y la sumatoria de la complejidad de este ciclo para las demás partes sería igual a $T(k, n) = (k * log(n))$, provocando un total de $T(k, m, n) = (k * (m - 1) * log(n) + k * (m - 1)) = (m * k * log(n) + 2 * k * log(n))$.

El segundo ciclo se encarga de almacenar cada una de las escenas de la parte en cuestión en nuevo arreglo, a partir del valor de sus grandezas totales. El iterador de este ciclo varía desde cero hasta el tamaño de la parte analizada y obtener la grandeza total de una escena es de tiempo logarítmico. Por lo tanto, la complejidad para el caso de la apertura es igual a $T(k, m, n) = (k * (m - 1) * log(n))$ y la sumatoria de las demás partes, $T(k, n) = (k * log(n))$, y en total $T(k, m, n) = (k * (m - 1) * log(n) + k * log(n))$.

El tercer ciclo se se encarga de recorrer el nuevo arreglo con los índices de las escenas organizadas por grandeza. Este contiene un ciclo for anidado que se encarga de obtener las escenas almacenadas con igual grandeza y almacenarlas en un nuevo arreglo, que posteriormente será insertado en posición correspondiente en el show. El ciclo exterior varía desde 6, que es la menor grandeza posible en cualquier instancia del problema, hasta biggestPartSceneAwesomeness, el valor de la mayor grandeza total de las escenas, en la parte analizada en el momento. Es posible hallar el máximo valor que esta variable puede adoptar, considerando una escena conformada por los tres animales de mayor grandeza, siendo igual a $n+(n-1)+(n-2) = 3*n-3$. Siendo $3*n-3-6+n = 3*n-9$ el número total de iteraciones. El for interno varía desde 0 hasta la cantidad de escenas con el mismo valor de grandeza. En el peor de los casos, este sería igual a n, dejando que valga 0 para todas las demás posiciones. Con esta información es posible encontrar la complejidad total de este ciclo, siendo igual a $T(3*n-9+n) = T(4*n-9)$.

Dado que el ciclo externo se repite m veces, la complejidad total de esta función es igual a $T(m,k,n) = (m*(m*k*log(n)+2*k*log(n)+k*(m-1)*log(n)+k*log(n)+4*n-9) = (m^2*k*log(n)+2*m*k*log(n)+m*k*(m-1)*log(n)+m*k*log(n)+4*n*m-9*m)$. Dado que el término dominante en esta expresión es $m^2*k*log(n)$, y tomando en cuenta los valores máximos m y k pueden adoptar, siendo estos 60 y n, respectivamente, la complejidad de esta función viene dada por $T(n) = (60^2*n*log(n))$, igual a $O(n*log(n))$.

```
void sortScenes() {

        for(int i=0; i<m; i++) {

                vector<vector<string>> part = show[i];
                int biggestPartSceneAwesomeness = 0;
                int partSize = part.size();
                for(int j=0; j<partSize; j++) {
                        int sceneAwesomeness = getSceneAwesomeness(part[j]);
                        if(sceneAwesomeness > biggestPartSceneAwesomeness) {

                                biggestPartSceneAwesomeness = sceneAwesomeness;
                                if(sceneAwesomeness > biggestSceneAwesomeness)
                                        biggestSceneAwesomeness = sceneAwesomeness;
                        }
                }

                vector<int> sortedIndexes[biggestPartSceneAwesomeness + 1];

                for(int j=0; j<partSize; j++) {

                        vector<string> scene = part[j];
                        int awesomeness = getSceneAwesomeness(scene);
                        sortedIndexes[awesomeness].push_back(j);
                }
```

```
                vector<vector<string>> sortedPart;
                for(int j=6; j<=biggestPartSceneAwesomeness; j++) {
                        vector<int> scenesWithEqualAwesomeness = sortedIndexes[j];

                        for(int l=0; l<scenesWithEqualAwesomeness.size(); l++) {
                                sortedPart.push_back(part[scenesWithEqualAwesomeness[l]]);
                        }
                }
                show[i] = sortedPart;
        }
}
```

sortParts: esta función se encarga de organizar todas las partes del show por su grandeza total, mediante el algoritmo de ordenamiento counting sort. Está compuesta por tres ciclos for que se analizarán a continuación.

Como ya se explicó, en la función sortScenes, el algoritmo counting sort almacenará los índices de las partes del show en una posición del arreglo asociado a la mayor grandeza total de una parte.

El primer ciclo va desde cero hasta m, el número total de partes, y se encarga de encontrar el mayor valor de grandeza que puede tener una parte, para así poder crear el arreglo que almacenará los índices de las partes. Además, suma a la variable showAwesomeness la grandeza de cada parte, para así encontrar la grandeza total del show. Puesto que hallar la grandeza de una parte tiene una complejidad de T(n) = n*log(n) y el mayor valor que puede alcanzar m es 60, la complejidad total de este ciclo es T(n) = 60*n*log(n), y tendiendo n al infinito, O(n*log(n)).

El segundo ciclo parte de 1 y va hasta m, excluyendo así la apertura del show, ya que se organizarán los índices de las demás partes dada su grandeza. La complejidad de este ciclo es constante, pues solo hay accesos a arreglos y vectores.

```
void sortParts() {

        int partAwesomeness[m];
        int biggestPartAwesomeness = 0;

        for(int i=0; i<m; i++) {

                int awesomeness = getPartAwesomeness(show[i]);
                partAwesomeness[i] = awesomeness;
                showAwesomeness += awesomeness;

                if(awesomeness > biggestPartAwesomeness)
                        biggestPartAwesomeness = awesomeness;
        }
```

```
        vector<int> sortedIndexes[biggestPartAwesomeness+1];

        for(int i=1; i<m; i++) {

                vector<vector<string>> part = show[i];
                int awesomeness = partAwesomeness[i];

                sortedIndexes[awesomeness].push_back(i);
        }

        vector<vector<vector<string>>> sortedShow;
        sortedShow.push_back(show[0]);

        for(int i=6; i<=biggestPartAwesomeness; i++) {

                vector<int> partsWithEqualAwesomeness = sortedIndexes[i];

                for(int j=0; j<partsWithEqualAwesomeness.size(); j++) {
                        sortedShow.push_back(show[partsWithEqualAwesomeness[j]]);
                }
        }

        show = sortedShow;
}
```

1.4.3 Solución O(n)

Puesto que la estructura de datos usada para a variable animals fue un unordered map, se regresa a lo que se había planteado al inicio del documento, en que se considerarán los accesos como una operación de complejidad constante.

Las funciones sortScenes y sortParts utilizadas en este implementación, son las mismas que se usan en la implementación de complejidad O(nlogn), presentado variaciones en la complejidad únicamente a la hora de acceder a las grandezas totales de las escenas o de las partes, provocando que cada uno de estos accesos que se realizaban en complejidad logarítmica, ahora se realicen en complejidad constante, permitiendo así que estas funciones, cuyo complejidad es O(nlogn), sea simplemente O(n).

Las demás funciones utilizadas, trabajan de la misma forma que las funciones de la implementación de complejidad $O(n^2)$, puesto que ya tenían una complejidad lineal y no era necesario modificarlas.

1.4.4 Script testCreator.py

Para generar pruebas con grandes cantidades de datos, se programó en python un generador de pruebas, que nos permitía especificar el nombre del archivo de prueba y los valores de las variables n, m y k. Este programa verifica que los datos ingresados sean correctos y produce un archivo de texto, que almacena en la misma carpeta que esté situado el programa.

Para saber cómo utilizar este programa, basta con escribir python3 testCreator.py -h en la terminal, estando dentro de la misma carpeta que esté el programa. Este se encuentra dentro del directorio *Pruebas*.

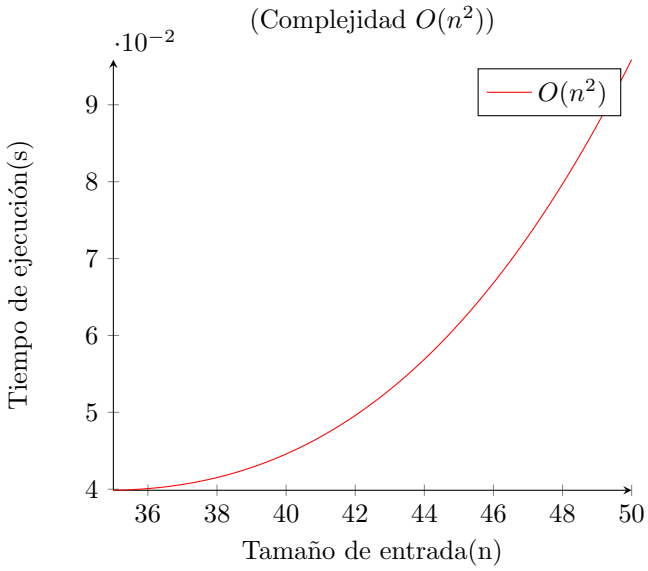
2. Análisis de resultados

2.1. Tamaño de entrada vs tiempo de salida

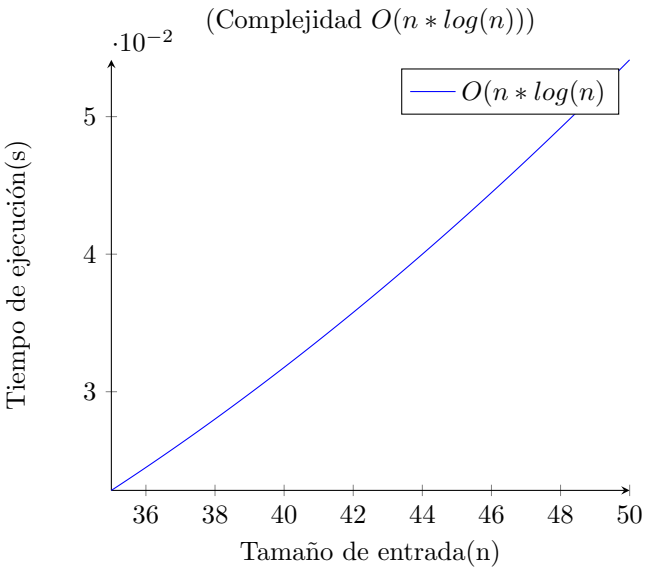
Ahora, se observan las gráficas de tamaño de entrada vs tiempo de salida, de cada complejidad individual de los algoritmos y de todas comparadas entre si.

Para realizar estas graficas se crearon distintas instancias de los problemas con un script en python y se midió el tiempo de ejecución de cada algoritmo con dichas instancias. Luego, se interpolaron los datos de forma lineal y cuadrática para producir las curvas.

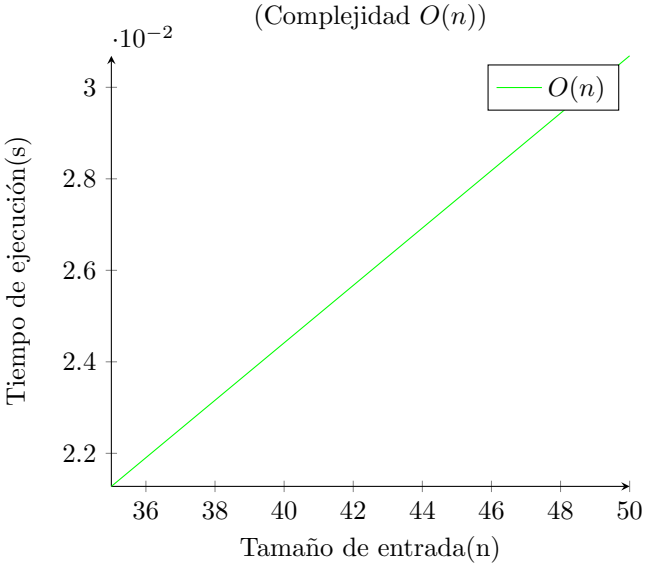
n	6	10	50	100	300
t(s)	0.00029833	0.00040666	0.09903266	2.54526216	139.32268



n	6	10	50	100
t(s)	0.00037916	0.00049083	0.05413383	0.104337

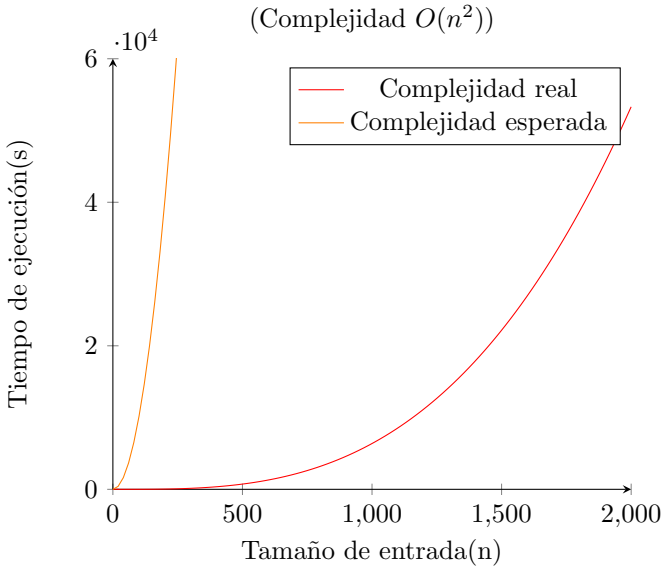


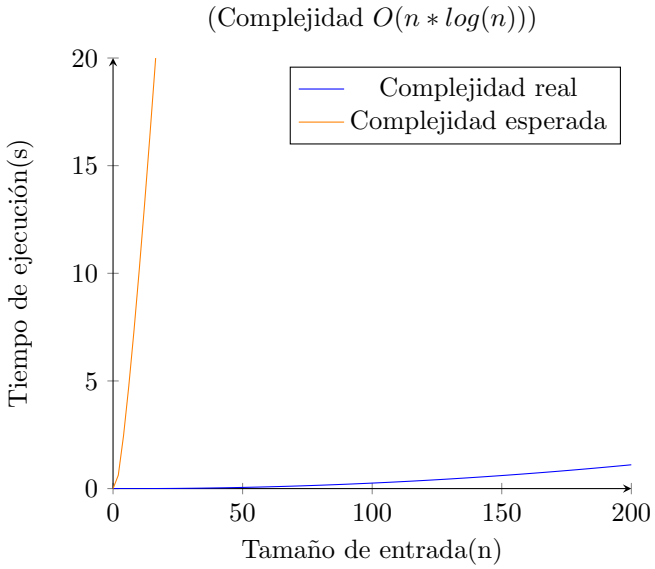
n	6	10	50	100
t(s)	0.00031883	0.000418	0.045191	0.05549633

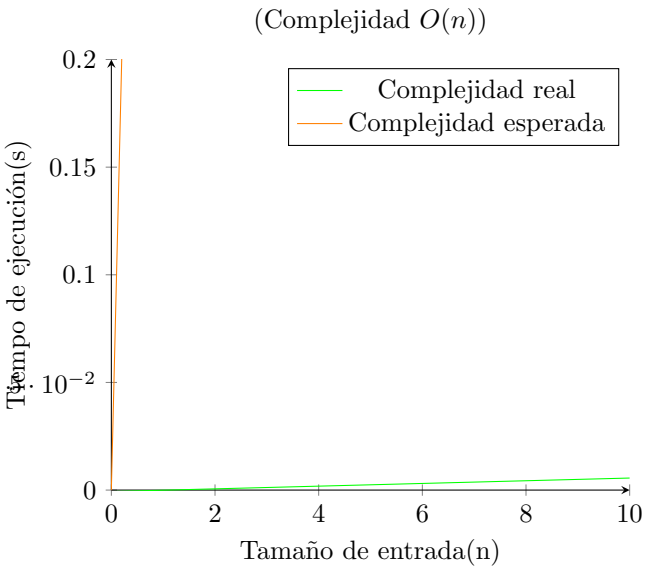


2.2. Complejidad teórica esperada vs complejidad real del algoritmo

A continuación se observa la comparación del tiempo esperado con complejidad real de cada algoritmo realizado para el proyecto.

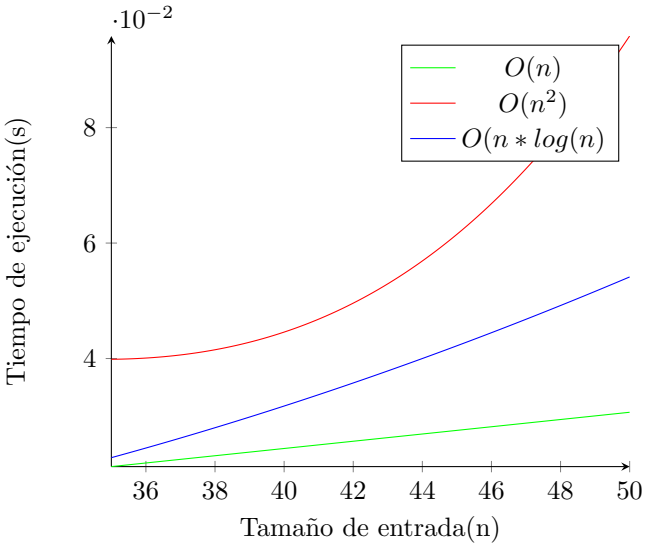





(Complejidad $O(n)$)
Complejidad real
Complejidad esperada
Tiempo de ejecución(s)
0.2
0.15
0.1
$\cdot 10^{-2}$
3
0
0
2
4
6
8
10
Tamaño de entrada(n)

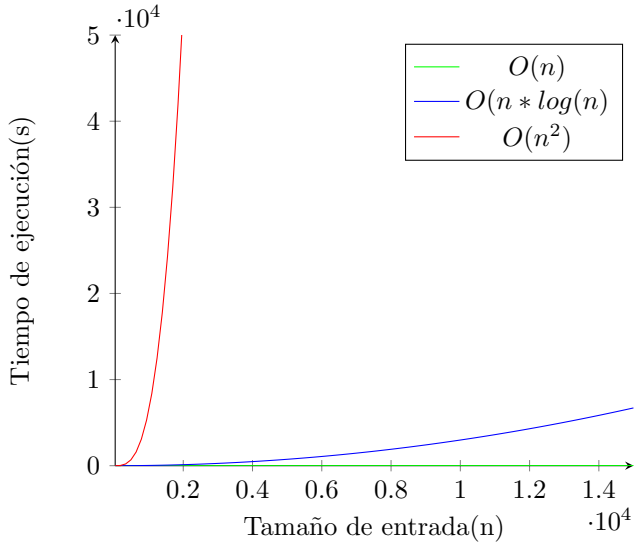
2.3. Comparación del desempeño de los algoritmos implementados

Complejidades comparadas con valores de entrada pequeños



Complejidades comparadas con valores de entrada grandes

Al intentar que se aprecien todas las gráficas, la pendiente de la función lineal es apenas visible



3. Conclusiones

Teniendo en cuenta los resultados evidenciados en las gráficas y las diferencias entre los algoritmos que no pueden ser apreciadas tan fácilmente, como el cambio de los algoritmos de ordenación entre la soluciones $O(n)$ y $O(n^2)$, y el cambio de una estructura de datos entre la soluciones $O(n)$ y $O(nlogn)$, se puede evidenciar que hay múltiples formas de implementar un algoritmo, pero aunque estas se vean relativamente parecidas, hay que analizar más a fondo las implementaciones de las estructuras de datos que se están usando y de las posibles variaciones de los métodos usados, para así usar el que mejor se adapte al problema y poder optimizarlo al máximo. A pesar de que se intentó programar un solución al desempate de las escenas con igual grandeza, fue imposible conseguir una complejidad lineal del mismo algoritmo, por lo que tampoco se pudo obtener la escena de menor y mayor grandeza total, puesto que se planeaban obtener con la misma función