Proyecto Final

Estudiantes:
Juan Sebastian Getial Getial - 2124644
Sebastian Idrobo Avirama - 2122637
Deison Aleixer Cardona Arias - 1840261
Jose Luis Hincapié Bucheli - 2125340

Asignatura: Análisis y Diseño de Algoritmos I

Docente: Jesus Alexander Aranda Bueno

Universidad del Valle
Santiago de Cali
2023

Desarrollo

Para la realización del proyecto, se decidió usar como base principal el algoritmo Counting-Sort, ya que este presenta una complejidad computacional teórica $O(n)$, esto debido a que se quiere realizar una solución general para el problema que sea $O(n)$, y por consiguiente $O(n * log(n))$ y $O(n^2)$. Consigo con el algoritmo base, el recorrido de estructuras tales como diccionarios o listas se hizo con ciclos for no anidados, logrando así que dicho recorrido escalara en una complejidad no mayor a $O(n)$

La forma en cómo se decidió representar los parámetro de entrada fue de la siguiente manera de problema fue de la siguiente manera: Las escenas son listas con los nombres de los animales que participan en cada escena y dichas escenas forman una parte (Esta puede ser la apertura o las demás partes que se componen), la cual es una lista de escenas. Los animales y sus grandezas se representaron a través de un diccionario con el nombre del animal como la llave y el valor como su grandeza.

A continuación, se hará una explicación de algunos de los algoritmos usados, esto debido a la similitud que existe entre estos teniendo en cuenta los puntos descritos anteriormente.

Para empezar, se utilizaron algunas funciones sencillas para obtener el máximo o mínimo valor de una lista de números, las cuales fueron **find_max_value** y **find_min_value**, que presentan una estructura idéntica:

```
1  def find_max_value(lst):
2      if not lst:
3          return None
4
5      max_value = lst[0]
6
7      for num in lst:
8          if num > max_value:
9              max_value = num
10
11     return max_value
```

```
1  def find_min_value(lst):
2      if not lst:
3          return None
4
5      min_value = lst[0]
6
7      for num in lst:
8          if num < min_value:
9              min_value = num
10
11     return min_value
```

Ambos algoritmos reciben la lista de valores y comparten una misma estructura, con única diferencia en la línea 8 la cuál define qué valor se requiere encontrar en la lista, si el mínimo o máximo. Si se toma como referencia la línea 8 para calcular el grado de complejidad de ambos algoritmos, es posible observar que presentan una complejidad $O(n)$, ya que la línea 8 se ejecuta debido al ciclo for de la línea 7 que recorre cada elemento de la lista, n veces siempre, siendo n el número de elementos de la lista de entrada.

Otro algoritmo utilizado para la grandeza total de una escena, su grandeza máxima y contabilizar la aparición de los animales, fue **grandeza_total_y_max**, el cual recibe una escena y las grandezas de cada animal:

```
1  def grandeza_total_y_max(escena, grandezas):
2      animal0 = escena[0]
3      count_animals[animal0] += 1
4      grandeza_total = grandezas[animal0]
5      grandeza_max = grandezas[animal0]
6      for i in range(1, len(escena)):
7          animal1 = escena[i]
8          count_animals[animal1] += 1
9          grandeza_total += grandezas[animal1]
10         if grandezas[animal1] > grandezas[animal0]:
11             grandeza_max = grandezas[animal1]
12             animal0 = animal1
13     return (grandeza_total, grandeza_max)
```

Se sabe que las operaciones de asignación son todas de valor constante, por lo que no se tienen en cuenta para realizar el análisis de complejidad. Por lo tanto, si se toma como referencia la línea 10, dicha línea se ejecutará, debido al ciclo for que se ejecuta una vez por cada animal de la escena exceptuando al primero, $n - 1$ veces, donde n es el número de animales de la escena.
No obstante, debido a la naturaleza del problema, se sabe que el número de animales de una escena siempre será 3, por lo que dicha línea se ejecutará una total de 2 veces siempre, lo cual significa que este algoritmo presenta una complejidad constante, es decir $O(1)$. Es importante resaltar, que la aparición de cada animal se va contabilizando en la línea 3 y 8, en un diccionario externo.

Como anteriormente fue mencionado, se utilizó el algoritmo de Counting-sort como algoritmo base para solucionar el problema, por lo que varios de nuestros algoritmos se basan en él para organizar los animales, las escenas y las partes, tales como **counting_sort**, **counting_sort_parts** y **counting_sort_scenes**. Debido a que ambos presentan prácticamente la misma estructura, solo se explicará uno de ellos, en este caso el **counting_sort_scenes**, y para ello se toma como referencia el pseudocódigo del Counting-Sort:

```
COUNTING-SORT(A,B,k)
 for i←1 to k
   do C[i]←0
 for j←1 to length[A]
   do C[A[j]]←C[A[j]]+1
 for j←2 to k
   do C[i]←C[i] + C[i-1]
 for j←length[A] downto 1
   do B[C[A[j]]]←A[j]
      C[A[j]]←C[A[j]]-1
```

Analice la complejidad

O(k)

O(n)

O(k)

O(n)

T(n)=O(2n+2k), como k=O(n)

T(n)=O(n)

```python
def counting_sort_scenes(array, grandezas_totales):
    size = len(array) # Numero de Escenas
    output = [0] * size
    max_index=find_max_value(grandezas_totales) + 1
    # Initialize count array
    count = [0] * max_index  #URGENTE CAMBIAR ESTO

    # Store the count of each elements in count array
    for i in range(0, size):
        count[array[i][1]] += 1

    # Store the cummulative count
    for i in range(1, max_index):
        count[i] += count[i - 1]

    # Find the index of each element of the original array in count array
    # place the elements in output array
    i = size - 1
    while i >= 0:
        escena = array[i]
        grandeza_total = escena[1]
        output[count[grandeza_total] - 1] = escena[0]
        count[grandeza_total] -= 1
        i -= 1

    #return output
    # Copy the sorted elements into original array
    for i in range(0, size):
        array[i] = output[i]
```

O(k)

O(n)

O(k)

O(n)

O(n)

C**ounting_sort_scenes** es una modificación del Counting-Sort convencional con diferencias más notorias en las primeras y últimas líneas. En el caso de las primeras líneas, se realizan algunos cálculos, como hallar la cantidad de escenas o inicializar algunos arrays, pero al final dichas primeras siguen teniendo una complejidad de $O(k)$, donde k es el valor máximo de las escenas, es decir, la grandeza total máxima de una escena. Y en el caso de las últimas líneas, es posible observar que estas no tienen equivalencia con ninguna parte del pseudocódigo, sin embargo, su complejidad es $O(n)$, donde n es el número de escenas.

Las demás líneas son equivalentes a las del pseudocódigo del Counting-Sort convencional, por lo que, al calcular la complejidad de este algoritmo, se obtiene como resultado $O(3n + 2k)$, pero como $k = O(n)$, finalmente el algoritmo tiene una complejidad $O(n)$.

Los demás algoritmos basados en el Counting-Sort tienen una estructura similar, por lo que al analizar su complejidad pueden haber ligeras diferencias, sin embargo, todos presentan una complejidad $O(n)$.

Teniendo en cuenta estos algoritmos, se crearon otros algoritmos que se encargan de aplicarlos a cada parte y sus escenas correspondientes. Entre dichos algoritmos se encuentran **ordenar_parte**, **ordenar_partes** y **ordenar_apertura**. Dichos algoritmos presentan una estructura análoga entre sí, pero realizan algunas acciones básicas diferentes con el objetivo de calcular a su vez otros aspectos como la escena con mayor o menor grandeza. Debido a esto y por no extender demasiado el documento, se explicara solo en este caso **ordenar_parte**:

```
1  def ordenar_parte(parte):
2      escenas_Ordenadas_Internas = []
3      grandezas_totales = []
4      grandezas_max = []
5      grandeza_total_escena = 0
6      for escena in parte:
7          (escena_ordenada, grandeza_total, grandeza_max) = counting_sort(escena, ANIMALES)
8          escenas_Ordenadas_Internas.append((escena_ordenada, grandeza_total, grandeza_max))
9          grandezas_totales.append(grandeza_total)
10         grandezas_max.append(grandeza_max)
11         grandeza_total_escena += grandeza_total
12
13     counting_sort_scenes_grandeza_max(escenas_Ordenadas_Internas, grandezas_max)
14     counting_sort_scenes(escenas_Ordenadas_Internas, grandezas_totales)
15     return [escenas_Ordenadas_Internas, grandeza_total_escena]
```

Gran parte del algoritmo anterior realiza operaciones básicas, por lo que en este caso solo tendremos en cuenta la línea 7, 13 y 14 para analizar la complejidad del mismo. Se puede apreciar que la línea 7 se ejecutará una vez por cada escena y, como fue explicado anteriormente, dicha línea tiene un costo lineal $O(p)$ debido al uso de un algoritmo basado en el Counting-Sort. De la misma manera, las líneas 13 y 14 tienen un costo lineal de $O(k)$. Por lo que la complejidad del algoritmo sería $O(kp + 2k)$, sin embargo, en este caso específico debido a la naturaleza del problema del zoológico, p tiene un valor de

3, ya que corresponde al número de animales por escena, por lo que la complejidad de este algoritmo sería finalmente $O(k)$, siendo k el número de escenas de la parte dada.

Al realizar un análisis similar para los 2 algoritmos restantes, es decir, **ordenar_apertura** y **ordenar_partes**, se obtienen los siguientes resultados:

- **ordenar_partes** = $O(t(k))$, donde t es el número de partes a ordenar y k el número de escenas de cada parte.
- **ordenar_apertura** = $O(a)$, donde a es el número de escenas de la apertura.

Para el análisis final, cabe resaltar que en la solución final se pueden observar algunos ciclos for extra, sin embargo, dichos ciclos son lineales. Aun así, se tendrán en cuenta para el análisis final. También como fue observado, todos los algoritmos usados en la solución final del problema son lineales con respecto a sus valores de entrada, por lo que al final es posible concluir que toda la solución es lineal, ya que todos los algoritmos usados son $O(n)$, donde n corresponde al valor de entrada correspondiente a cada algoritmo.

Finalmente, si se desea una solución final más detallada, se observará que presenta una complejidad igual $O(tk + a + 2n + (m - 1))$, pero, debido a la naturaleza del problema, se sabe que $t = m - 1$ y $a = (m - 1)k$.

Por lo que la complejidad sería:
$O((m - 1)k + (m - 1)k + 2n + (m - 1)) = O((m - 1)(2k + 1) + 2n)$

Análisis de Resultados

Teniendo en cuenta que la cota superior designada para el algoritmo fue $O(n)$, este resultado debe de verse reflejado sobre los tiempos de operación de la función principal (**solution)**, siendo que al tomar múltiples tiempos de ejecución del algoritmo con diferentes tamaños de entrada, se debe de reflejar una función que obedece un orden lineal.

Para el respectivo análisis se procedió a evaluar la función dividido en tres partes:

- El cambio del tiempo de ejecución respecto al número de escenas en una parte (K)
- El cambio del tiempo de ejecución respecto al número de partes (M)
- El cambio del tiempo de ejecución respecto al número de animales (N)

Esto debido a que la función principal opera sobre tres múltiples variables, por lo que el cambio constante de una de estas variables no debe superar una cota superior a $O(n)$. También se tiene en cuenta que el cambio de una variable afectará los tiempos de ejecución, por lo que se trabaja dividdamente con un cambio de dato constante sobre una sola variable para visualizar los resultados de una manera clara[1]. Téngase en cuenta que si el cambio del tiempo respecto a K, M y N demuestran no superar $O(n)$ separadamente, significa por entonces que la función en su conjunto obedecerá a una función lineal, respetando la cota superior designada.

Teniendo en cuenta lo anteriormente expuesto, al realizar una toma de tiempos con un cambio constante en los datos de entrada, se obtuvieron los siguientes resultados:

Respecto al número de escenas en una parte (K)

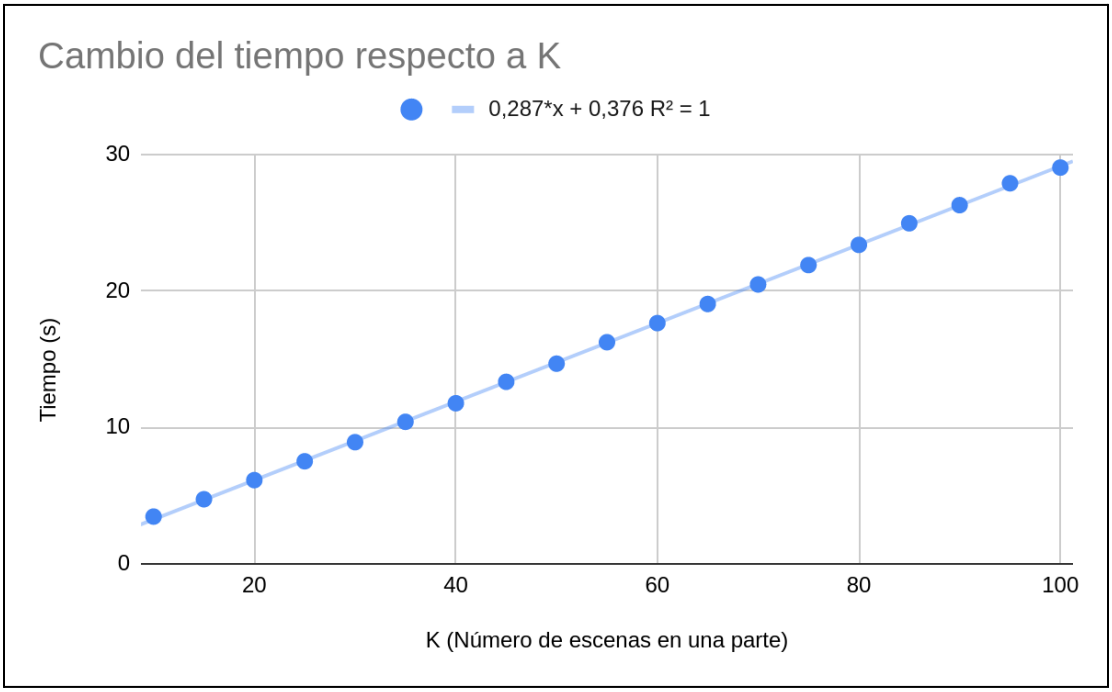
Para la medición de la función en base a la variable K se procedió a variar la variable adicionando 5 números de escenas por tiempo, obteniendo así los siguientes datos:

Parámetros de entrada	
n	1000
m	1000
k	**Tiempos (s)**
10	3,46
15	4,73
20	6,13
25	7,51
30	8,91
35	10,4

[1] El archivo con el que se realizó el respectivo análisis fue con *analisis.py*

40	11,76
45	13,33
50	14,66
55	16,23
60	17,63
65	19,03
70	20,46
75	21,88
80	23,36
85	24,94
90	26,27
95	27,87
100	29,02

Introduciendo los datos en un gráfico de dispersión, y agregando una línea de tendencia que refleje el comportamiento de la respectiva función, se obtiene el siguiente gráfico:



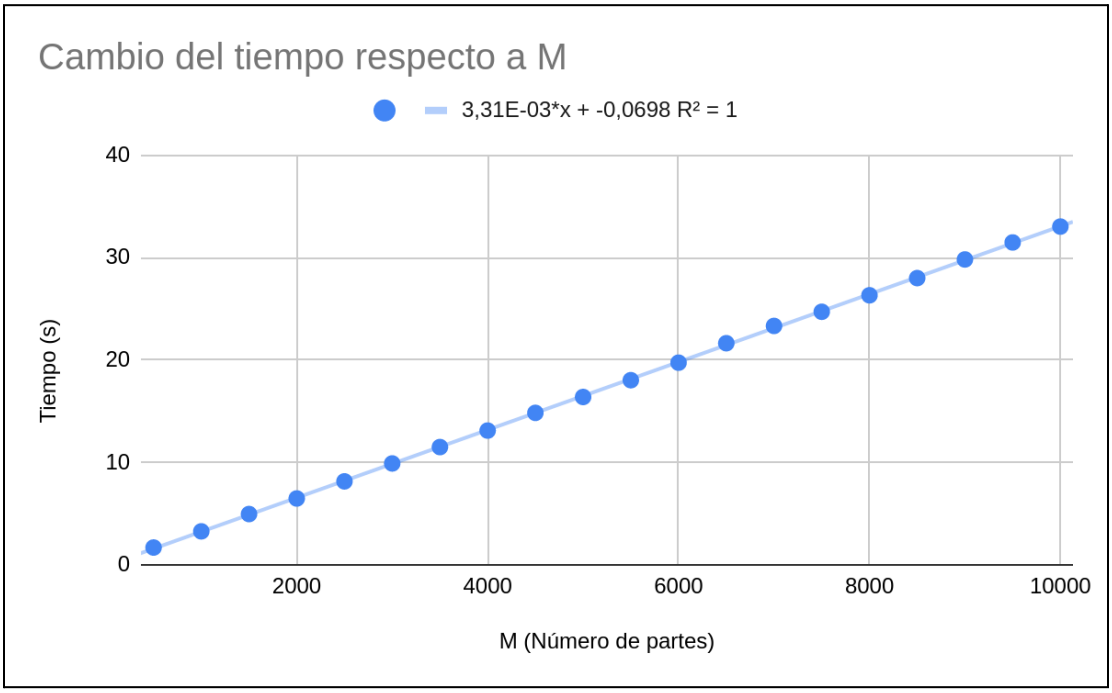
Por lo que al observar el cambio del tiempo respecto a K, y consigo el comportamiento de la línea de tendencia, es posible afirmar que esta obedece a una función lineal que se encuentra acotada por $O(n)$.

Respecto al número de partes (M)

Para la medición de la función en base a la variable *M* se procedió a variar la variable adicionando 500 partes, obteniendo así los siguientes datos:

Parámetros de entrada	
n	1000
k	10
m	**Tiempos (s)**
500	1,69
1000	3,27
1500	4,96
2000	6,48
2500	8,15
3000	9,9
3500	11,5
4000	13,11
4500	14,84
5000	16,39
5500	18,03
6000	19,74
6500	21,64
7000	23,33
7500	24,71
8000	26,32
8500	28
9000	29,82
9500	31,48
10000	33,03

Introduciendo los datos en un gráfico de dispersión, y agregando una línea de tendencia que refleje el comportamiento de la respectiva función, se obtiene el siguiente gráfico:



Por lo que al observar el cambio del tiempo respecto a M, y consigo el comportamiento de la línea de tendencia, es posible afirmar que esta obedece a una función lineal que se encuentra acotada por $O(n)$.

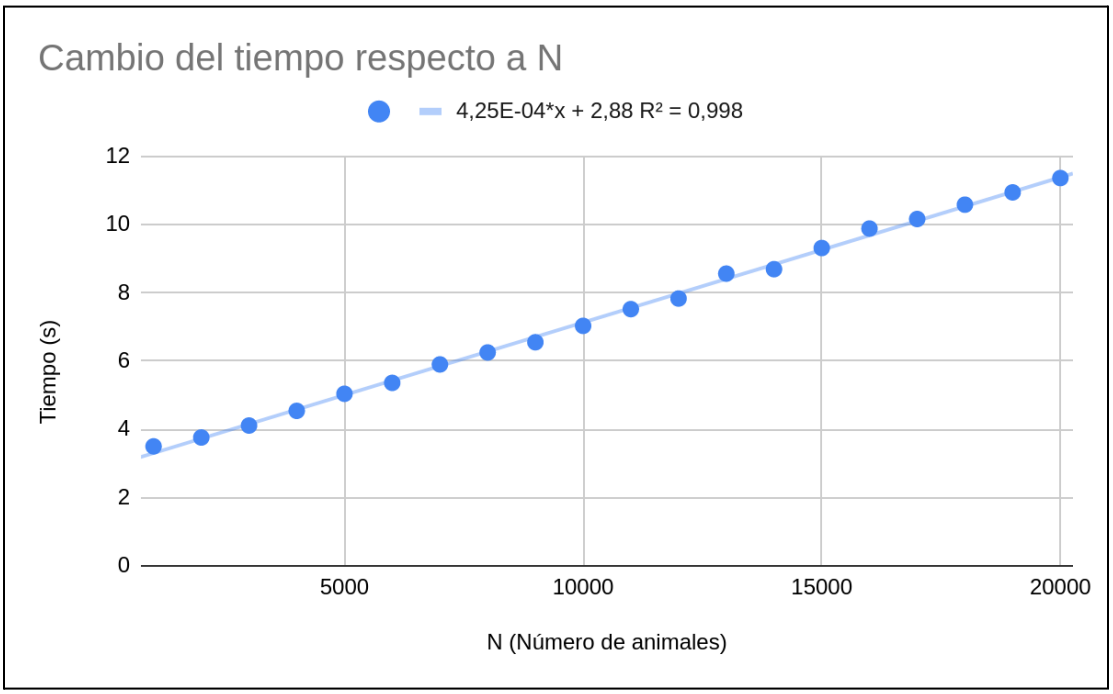
Respecto al número de animales (N)

Para la medición de la función en base a la variable N se procedió a variar la variable adicionando 1000 animales, obteniendo así los siguientes datos:

Parámetros de entrada	
m	1000
k	10
n	**Tiempos (s)**
1000	3,5
2000	3,76
3000	4,11
4000	4,54
5000	5,04
6000	5,36
7000	5,9
8000	6,25
9000	6,55
10000	7,03

11000	7,52
12000	7,83
13000	8,56
14000	8,69
15000	9,31
16000	9,88
17000	10,16
18000	10,58
19000	10,94
20000	11,36

Introduciendo los datos en un gráfico de dispersión, y agregando una línea de tendencia que refleje el comportamiento de la respectiva función, se obtiene el siguiente gráfico:



Por lo que al observar el cambio del tiempo respecto a N, y consigo el comportamiento de la línea de tendencia, es posible afirmar que esta obedece a una función lineal que se encuentra acotada por $O(n)$.

Conclusiones

- Debido a que la solución planteada tiene una complejidad igual a $O(n)$, se puede decir que dicho solución también presenta una complejidad igual a $O(n^2)$ y $O(n * log(n))$, ya que al estar acotada por n, también está acotada por n^2 y $n * log(n)$. Sin embargo, la complejidad más precisa es $O(n)$.
- El desarrollo de algoritmos eficientes resulta crucial para lograr una solución óptima a un problema, ya que si bien existen múltiples enfoques para resolver un problema, la eficiencia de los mismos puede variar significativamente. Es por esto que resulta fundamental realizar un correcto análisis, ya que de esto dependerá si obtenemos un algoritmo cuyo desempeño sea excelente, o pésimo.
- Las cotas superiores asintóticas, y consigo la notación *O grande* no solo son términos que se expresan teóricamente, sino que la exposición de las mismas también es posible encontrarlas prácticamente, siendo la demostración de las mismas expresadas en el análisis de resultados a través de los tiempos computacionales obtenidos de los algoritmos, cuyos resultados fueron expresados en una serie de datos que fueron expresados en una gráfica de orden lineal, como fue concluido en el desarrollo.