Miniproyecto

Sebastián Betancourt   Wilmer Bautista   Javier Diaz

Análisis

La tarea planteada es un problema de ordenamiento un tanto ofuscado.

Para ordenar los animales dentro de una escena, el script utiliza un algoritmo de ordenamiento *naive* que aprovechando el tamaño fijo de la escena compara la grandeza del primer animal con la del segundo, y la del más pequeño con la del tercero, y de acuerdo al resultado puede necesitar comparar el tercero con el primero para concluir cual es el orden de los tres animales. Como se realizan a lo mucho 3 comparaciones, ordenar los animales en una escena requiere $3 \in O(1)$ pasos.

Para los demás ordenamientos, la solución planteada usa uno de los tres algoritmos de ordenamiento disponibles: CountingSort, MergeSort e InsertionSort, de acuerdo al argumento que se le pase el usuario. Con los dos algoritmos de los anteriores párrafos, el script `proy.py` hace lo siguientes pasos:

1. Ordenar los animales en las escenas de la apertura. Cuando se ordenan los animales de una escena, también se registra:
   (a) Si la escena en cuestion es de mayor grandeza total a la escena registrada en la variable global `escenaMasGrande`. De serlo, la escena es cuestión se almacena en `escenaMasGrande`.
   (b) Si la escena en cuestion es de menor grandeza total a la escena registrada en la variable global `escenaMasPequena`. De serlo, la escena es cuestión se almacena en `escenaMasPequena`.
   (c) El aporte ponderado de la grandeza de esta escena al promedio total, que es actualizado iterativamente de la forma $promedio := promedio + \frac{grandezaEscena}{totalEscenas}$.
   (d) La aparición de los tres animales en un diccionario global `aprcns` que lleva cuenta de la cantidad de apariciones en escenas de cada animal.
2. Ordernar las escenas de la apertura de acuerdo a su máxima grandeza individual (como fue definido en el documento)
3. Ordernar las escenas de la apertura de acuerdo a su grandeza total. Como los tres posibles algoritmos de ordenamiento son estables, el orden por máxima grandeza individual prevalece para las escenas de igual grandeza total.

4. Ordenar los animales en las escenas de cada una de las partes. Los chequeos hechos en el primer item se vuelven a hacer. Esto último es redundante puesto que las escenas de las partes son las mismas escenas de la apertura.
5. Ordernar las escenas de cada una de las partes de acuerdo a su máxima grandeza individual (como fue definido en el documento)
6. Ordernar las escenas de cada una de las partes de acuerdo a su grandeza total. El orden por máxima grandeza individual prevalece para las escenas de igual grandeza total.
7. Ordenar las partes posteriores a la apertura de acuerdo a su grandeza total (como fue definido en el documento)
8. Ordernar con CountingSort el diccionario `aprcns` según las apariciones para tomar los primeros elementos como los animales de menor aparición y los últimos elementos como los animales de menor aparición. Se usa CountingSort específicamente para que su complejidad no crezca más que el algoritmo seleccionado.
9. Imprimir cada cada animal de cada escena de la apertura
10. Imprimir cada cada animal de cada escena de cada parte posterior a la apertura
11. Imprimir todos los animales con mayor participación
12. Imprimir todos los animales con menor participación
13. Imprimir el promedio de grandeza de las escenas

Complejidad teórica

Cada uno de estos pasos puede depender del "tamaño" de la entrada. Este "tamaño" realmente puede depender de tres variables: n, que es la cantidad de animales diferentes, m que es la cantidad de partes, y k que es la cantidad de escenas en cada parte posterior a la apertura. La complejidad claramente depende también del algoritmo de ordenamiento que se use. En la siguiente tabla estimamos la complejidad teórica los pasos listados previamente.

Para su lectura, es bueno aclarar que:

- Definimos $Sort(z)$ como la complejidad del algoritmo de ordenamiento seleccionado, es decir, $O(zlogz)$ para MergeSort, $O(z^2)$ para InsertionSort y $O(z)$ para CountingSort (este último también depende de la cota, se profundizará más adelante).
- Ordenar los animales de una escena es de orden $O(1)$.
- Tomamos $3 \leq n$, $2 < m \leq 60$ y $1 \leq k \leq n$.
- Desconocemos los detalles de implementación de las estructuras de datos. Supondremos que las operaciones inserción, eliminación y acceso en cualquier posición no dependen del tamaño de la entrada del problema y por lo tanto son de complejidad $O(1)$. Para

relacionar los animales con sus grandezas o apariciones, por ejemplo, usamos diccionarios en Python que usan tablas de Hash y sus operaciones son en promedio $O(1)$.

- Operaciones auxiliares como impresión, actualización de globales o cálculo de grandeza de las escenas también se tomarán de costo constante.
- No pretendimos ser particularmente eficientes desarrollando la solución (empezando porque elegimos Python). Es problable que ordenar solo las partes posteriores y luego hacer algo parecido a Merge para construir la apertura ya organizada hubiera resultado más barato. Tampoco era necesario repetir los literales (1a) - (1d). Pretendemos escribir un código y analizarlo.

Paso	**Complejidad estimada**
1. Ordenar los animales en las escenas de la apertura. *Por cada escena*, también (1a) Actualizar `escenaMasGrande` (1b) Actualizar `escenaMasPequena` (1c) Actualizar promedio (1d) Actualizar `aprcns`	$(m-1)*k*O(1) = O(mk)$ $(m-1)*k*O(1) = O(mk)$ $(m-1)*k*O(1) = O(mk)$ $(m-1)*k*O(1) = O(mk)$ $(m-1)*k*O(1) = O(mk)$
2. Ordenar las escenas de la apertura de acuerdo a máxima grandeza individual	$Sort((m-1)*k) = Sort(mk)$
3. Ordenar las escenas de la apertura de acuerdo a su grandeza total.	$Sort((m-1)*k) = Sort(mk)$
4. Ordenar los animales en las escenas de cada una de las partes. (1a), (1b), (1c), (1d) se repiten	$(m-1)*k*O(1) = O(mk)$ $(m-1)*k*O(1) = O(mk)$
5. Ordernar las escenas *de cada una de las partes* de acuerdo a su máxima grandeza individual	$Sort((m-1)*k) = Sort(mk)$
6. Ordernar las escenas *de cada una de las partes* de acuerdo a su grandeza total.	$Sort((m-1)*k) = Sort(mk)$
7. Calcular la grandeza de cada parte posterior a la apertura y ordenarlas	$O(k)Sort(m-1) = Sort(mk)$
8. Ordernar con CountingSort el diccionario `aprcns` que registra las apariciones de cada animal	$O(n+mk)$: n animales que aparecieron máximo $2*(m-1)*k$ veces
9. Imprimir cada animal de cada escena de la apertura 10. Imprimir cada animal de cada escena de cada parte posterior a la apertura 11. Imprimir todos los animales con mayor participación 12. Imprimir todos los animales con menor participación 13. Imprimir el promedio de grandeza de las escenas	$3*(m-1)*k = O(mk)$ $3*(m-1)*k = O(mk)$ $O(n)$ $O(n)$ $O(1)$

Entonces, estimar la complejidad total del problema depende de la relación entre n, m y k. Mientras n se mantenga por debajo de $O(mk)$, podemos asegurar que cuando se usa InsertionSort la complejidad de todo el algoritmo es de $O((mk)^2)$, cuando se usa MergeSort, de $O(mk\log(mk))$, y cuando se usa CountingSort, de $O(mk)$. En general, mientras $O(n)$ se mantenga por debajo de $O(mk)$, la complejidad es $Sort(mk)$. Esta condición es explicitable, por ejemplo para CountingSort solo se debe cumplir que $n < 2mk - 2k$.

Pruebas

Metodología

Estas pruebas fueron ejecutadas en Python 3.7.7 (default, Mar 10 2020, 15:43:33) [Clang 11.0.0 (clang-1100.0.33.17)] en Windows 10 con un Intel Core i7-7700HQ a 2.8ghz y 16gb de RAM.

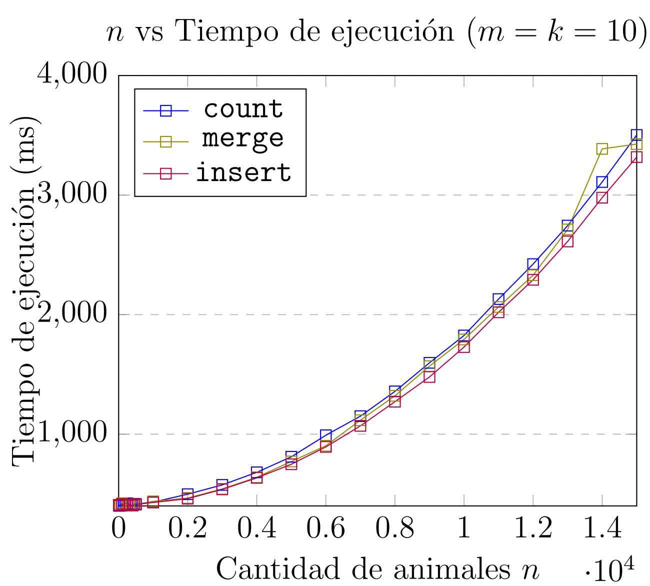
En PowerShell el comando `Measure-Command {python gen.py 5000 10 10 | python proy.py merge}` mide el tiempo de ejecución de generación y solución un problema con $n = 5000$, $m = 10$ y $n = 10$, con MergeSort, por ejemplo. La metología consisitió en variar estos tres parámetros con los tres algoritmos y registrar el tiempo de ejecución. Como las condiciones para n, m y k propuestas en el enunciado no permiten ver con claridad cambios notables en el tiempo de ejecución, jugamos con valores mucho más grandes.

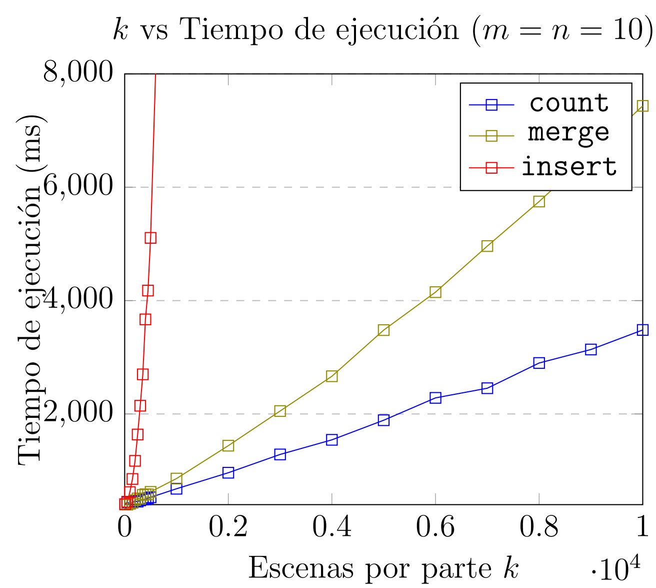
Para replicar el experimento y revisar el script, refiérase al documento `leeme.pdf` que contiene instrucciones detalladas de uso.

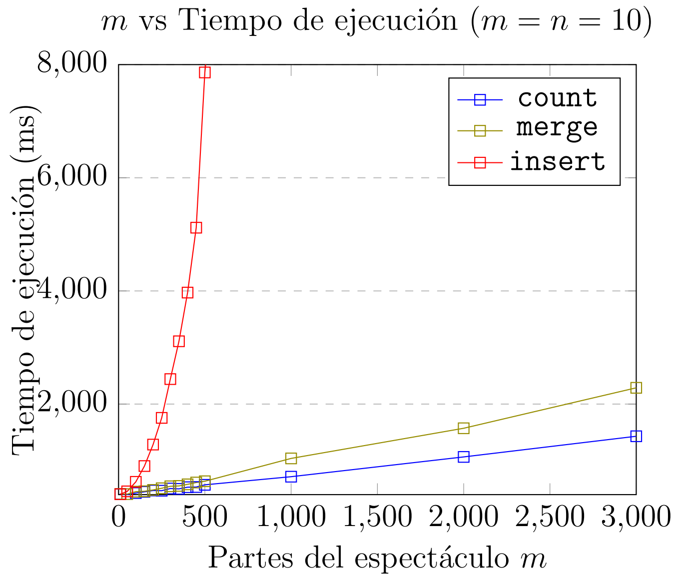
Resultados

Los datos brutos fueron registrados en la hoja de Excel `datos.xlsx` que se encuentra adjunta. Cuando se varió k y m, los tiempos de cómputo crecieron inúsitadamente, por lo que se tuvieron que suspender las pruebas para $k > 3000$ y $m > 3000$. Por curiosidad, dejamos correr una prueba con $k = 10000$, $m = n = 10$, y el tiempo de ejecución fue de 2425983 ms (más de 40 min). También hicimos una prueba con $m = 10000$, $k = n = 10$ y se demoró 2783891 ms (más de 45 min).

A continuación se muestran las gráficas comparando el rendimiento de los algoritmos variando n, m y k.







Conclusiones

Las graficas apoyan la hipótesis descrita en el análisis teórico: El tiempo de ejecución depende de m y k más que de n. En las pruebas variando n para valores $n < mk$, el tiempo realmente se mantiene constante, es solo para valores grandes de n que el tiempo empieza a depender de n. Usamos CountingSort para ordenar las grandezas en `aprcns`, Y este es el que más afectado se ve por el crecimiento de n (pues su cota depende directamente), mientras que los otros ordenamientos solo dependen de m y k que siempre son iguales a 10. El problema yace en la decisión de usar *siempre* CountingSort para ordenar aprcns, que termina demorándo más que los otros pasos. Por eso en n vs. tiempo de ejecución las tres variantes del script se comportan igual.

Hay una diferencia clara en las graficas que varian n con respecto a las que varían con k y m. En las últimas sí se nota una diferencia de acuerdo a algoritmo seleccionado. Aunque no describen tan claramente como quisieramos las funciones representativas $y = x$ o $y = x \log x$, sí hay una clara diferencia y una tendencia a seguir los valores descritos por estas funciones. Un buen ejercicio sería realizar una ajuste y mirar la efectividad con alguna prueba estadística como Kolmogórov-Smirnov.

Para dar una respuesta directa a la pregunta "¿Cuál es la compejidad del algoritmo?", los resultados apoyan que con `insert` es $O((mk)^2)$, con `merge` es $O(mk \log(mk))$, y con `count` es $O(mk)$. Ahora, para cumplir con los requerimientos del enunciado, sabemos que $m < 60$ y que $k < n$, entonces $Sort(mk) = Sort(60n) = Sort(n)$. Por lo tanto la complejidad también `insert` es $O(n^2)$, con `merge` es $O(n \log n)$, y con `count` es $O(n)$. Sin embargo creemos $Sort(mk)$ es una cota más justa.