Parcial III de estructuras de datos y algoritmos:

# Análisis de datos:

Los datos presentados para analizar representan características de automóviles. Hay 22 campos de distintos tipos que van a ser representados por una clase automóvil. Los campos son representados por los tipos, string, entero y float. Si bien a efectos de esta tarea contamos con una pequeña cantidad de datos, para pensar este problema vamos a suponer que estamos brindando soluciones a una multinacional gigantesca con un gran database.

# Análisis de algoritmos a implementar:

Dado que contamos con gran cantidad de datos, los algoritmos de orden cuadrático quedan totalmente descartados, dado que su ejecución para conjuntos grandes de datos tardaría cantidades de tiempo enormes.

Nuestra tarea consiste en ordenar el conjunto de automóviles por marca, luego dentro de cada marca por potencia y luego dentro de cada potencia por precio, por lo que de los 22 campos brindados solo vamos a considerar 3. Siempre tendremos que ordenar en forma ascendente

### Alternativa 1:

Podríamos considerar representar estos tres campos mediante su concatenación y aplicar un algoritmo de radixsort. De esta forma podríamos poner a precio como menos significativo y luego potencia y marca en este orden. De esta forma nuestro orden de tiempo de ejecución sería de O(N\*k). donde N es la cantidad de elementos y k es el largo de la concatenación de las claves.

### Alternativa 2:

La segunda alternativa sería aplicar un algoritmo de HeapSort, de forma tal que obtengamos un orden de tiempo de ejecución de O(Nlog(N)) en todos los casos cada vez que se requiera ordenar. Para esto tendríamos que definir la comparación entre instancias del tipo automóvil para que, si dos automóviles tienen la misma marca, se decida por su potencia y si tienen la misma potencia se decida la desigualdad por su precio.

### Alternativa 3:

Otro algoritmo famoso por su rendimiento es el QuickSort, este algoritmo tiene un caso promedio de O(Nlog(N)), para ello utiliza la técnica de algoritmos recursivos divide y vencerás. El peor caso del QuickSort supone que se elige el peor pivote posible y aquí es donde nos arriesgamos a que su orden pase a ser O(N2), pero, cuanto más grande es el conjunto la probabilidad de que esto pase decrece. Para implementar esta alternativa tendríamos que definir la comparación entre las instancias de automóvil como se describió en la alternativa anterior.

# Resolución:

Dado que el algoritmo RadixSort utiliza memoria extra para funcionar, aunque su orden es bastante bueno tenemos la posibilidad de usar QuickSort o HeapSort que no requieren memoria extra, por ende, RadixSort queda descartado y más aún cuando vamos a trabajar con bases de datos gigantescas como en este caso. Al ahorrar memoria estamos ahorrando mucha memoria que nunca viene mal.

QuickSort es un muy buen algoritmo de ordenación, pero es susceptible a mal funcionamiento en algunos casos, si nos envían un conjunto ordenado o medianamente ordenado hay probabilidad de que nos estalle la pila de ejecución y más si trabajamos con muchos datos. Incluso si no nos estalla la pila de ejecución hay probabilidad de que el algoritmo demore más de lo deseado. Entonces, como estamos trabajando con una gran empresa de renombre, no queremos dar posibilidad a cualquier error de funcionamiento, por esta razón vamos a implementar una solución utilizando HeapSort que es en promedio un poco más lento, pero es mucho más seguro.