1. A) Los más rápidos en el mejor de los casos son BubbleSort e InsertionSort. Ambos tienen una complejidad de O(n) si los datos ya están ordenados.

B) Inserción.

C) Los tres comparten la misma complejidad. La rapidez en sí podríamos delimitarla por cuál de los algoritmos tiene menos cantidad de comparaciones e intercambios (asignaciones). Inserción es un buen candidato a ser “rápido”, ya que tiene menos intercambios.

1. TODO: Shell, Cuenta de Distribución.
2. **Estable:** El orden relativo de las claves que sean iguales permanece invariante luego de haber sido ejecutado el algoritmo. Como regla general, aquellos algoritmos que hagan intercambios que no sean contiguos, no son estables.

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **Estable** | **Sensible** |
| Burbujeo | Sí | Sí |
| Inserción | Sí | Sí |
| Selección | No | No |
| QuickSort | No | No |
| MergeSort | Sí | No |
| ShellSort | No | Sí |
| CountingSort | Sí | No |

**Sensible:** El algoritmo es más o menos eficiente dependiendo de cómo esté la entrada (e.g. la entrada está ordenada, desordenada, al azar, etc.).

1. ^
2. .
3. TODO
4. Las dos características teóricas principales de los algoritmos de clasificación son la complejidad del tiempo y la complejidad del espacio.

En general, complejidad del tiempo nos permite saber cómo cambia el rendimiento del algoritmo a medida que aumenta el tamaño del conjunto de datos. Cosas para considerar:

* **¿Cuántos datos esperas ordenar?** Esto lo ayudará a saber si necesita buscar un algoritmo con una complejidad de tiempo muy baja.
* **¿Qué tan ordenados serán sus datos?** ¿Estará parcialmente ordenado? ¿Aleatoriamente ordenado? Esto puede afectar la complejidad del tiempo del algoritmo de clasificación. La mayoría de los algoritmos tendrán el peor y el mejor de los casos: desea asegurarse de que no esté utilizando un algoritmo en el conjunto de datos del peor de los casos.
* **La complejidad del tiempo no es lo mismo que el tiempo de ejecución.** Recuerde que la complejidad del tiempo solo describe cómo varía el rendimiento de un algoritmo a medida que aumenta el tamaño del conjunto de datos. Un algoritmo que siempre pasa por una sola vez sobre toda la entrada sería O (n) - su rendimiento se correlaciona linealmente con el tamaño de la entrada. Pero, un algoritmo que siempre hace dos pasadas sobre el conjunto de datos también es O (n) - la correlación sigue siendo lineal, incluso si la constante (y el tiempo real de ejecución) es diferente.

De forma similar, la complejidad del espacio describe la cantidad de espacio que necesita ejecutar un algoritmo. Por ejemplo, un tipo simple como tipo de inserción necesita una cantidad fija adicional de espacio para almacenar el valor del elemento que se está insertando actualmente. Esta es una complejidad de espacio auxiliar de O (1) - no cambia con el tamaño de la entrada. Sin embargo, tipo de fusión crea matrices adicionales en la memoria mientras se ejecuta, con una complejidad de espacio auxiliar de O (n). Esto significa que la cantidad de espacio adicional que requiere se correlaciona linealmente con el tamaño de la entrada.

Por supuesto, el diseño de algoritmo es a menudo una compensación entre el tiempo y el espacio: los algoritmos con una complejidad de espacio baja pueden requerir más tiempo, y los algoritmos con una complejidad de tiempo baja pueden requerir más espacio.

3. <https://stackoverflow.com/questions/70402/why-is-quicksort-better-than-mergesort>   
     
   En general, los dos son bastante buenos, pero tienen trade-offs que quizás los hagan mejores en situaciones bastante específicas. Por ejemplo, Merge Sort sería más eficiente cuando tengamos que ordenar datos cuya entrada provenga de un almacenamiento externo lento. QuickSort lee estos datos uno por uno en cada ciclo, mientras que Merge Sort hace una lectura secuencial de todos los datos que necesita para la partición.   
     
   QuickSort aprovecha bien el principio de localidad de la memoria caché, no necesita espacio extra en memoria, y con algunas mejoras, su worst-case puede ser mejorado para que sea igual que el de Merge Sort.   
     
   Si bien ambos son buenos, existen algoritmos más realistas, como TimSort, que se basan en asumir que, en el mundo real, en promedio, los datos se presentan con cierto orden. Existen variaciones basadas tanto en QuickSort como Merge Sort que son “inteligentes”, e intentan aprovecharse de esta suposición.

1. **Elegir un buen pivote:** una de las formas de mejorar QuickSort es a través de la elección de un pivote adecuado. Generalmente se elegía al último elemento del array como el pivote, pero esto causaba un worst-case con complejidad O(n2) para arrays que ya están ordenados.   
     
   Algunas alternativas para elegir un pivote incluyen elegir un elemento al azar del array, o hacer la mediana del primer, del último, y del elemento del medio del array. Sin embargo, la forma más robusta suele ser hacer la mediana de tres elementos al azar dentro del array.   
     
   **Hacer una partición triple:** la partición triple ayuda a no caer en un worst-case de O(n2) al momento de ordenar un array con muchos valores iguales.

**Combinar algoritmos:** otra de las formas más comunes de mejorar QS es a través del uso de otros algoritmos de ordenamiento cuando la entrada posee una cantidad de elementos baja. Generalmente, se usa el algoritmo de inserción para ordenar entradas cortas, y QS las demás.

1. Ni idea.
2. QS para listas largas, inserción para listas cortas. Inserción resulta más eficiente en ese caso porque no utiliza recursividad, es más estable, y requiere menos memoria que QS. Es por esta razón que el algoritmo de inserción es el caso base que utilizan tanto QS y Merge Sort en sus versiones mejoradas.
3. Shell Sort puede ser visto como una generalización del Insertion Sort. Shell Sort realiza comparaciones de elementos que están alejados, mientras que Insertion Sort hace comparaciones con elementos contiguos. Esto hace que Shell sea más rápido, pero en contraste que Insertion sea estable.

Insertion Sort tiene una complejidad de O(n2) y Shell Sort tiene una complejidad que ronda por O(n5/4).

1. No hay un algoritmo que sea el mejor en todas las situaciones, sin embargo, para uso en casos del mundo real, Timsort resulta ser uno de los mejores, y como el mismo está basado en MergeSort y el algoritmo de inserción, elijo a esos dos como los mejores.
2. La entrada ya estaba ordenada. Usar QS con las mejoras nombradas anteriormente (3-way partitioning) para no caer en ese escenario.
3. <https://www.toptal.com/developers/sorting-algorithms/nearly-sorted-initial-order>   
     
   El algoritmo de inserción resulta ser el más rápido.
4. El tiempo de ejecución varía mucho dependiendo de la entrada y del algoritmo utilizado. No hay ninguno que asegure realmente un tiempo de ejecución mínimo en todos los casos, ni QuickSort con todas sus posibles mejoras. Si hablamos en promedio, volvemos a la discusión de si utilizar QuickSort o MergeSort.
5. MergeSort, debido a que, al trabajar con una lista, el principio de localidad de la cache del que tanto se aprovecha QuickSort se ve totalmente anulado. Las direcciones de memoria de los elementos de la lista no tienen por qué ser contiguas, y esto causa que se reduzca la eficiencia en QS.   
     
   <https://stackoverflow.com/questions/7629904/why-is-mergesort-better-for-linked-lists>