Enunciado:

Resuelva los siguientes ejercicios sobre algoritmos de ordenamiento y de selección básicos. Analice el peor y el mejor casos para el tiempo de ejecución en los ejercicios que corresponda. Para realizar algunos de los ejercicios puede usar los siguientes tipos de dato: `std::vector<int>` o `std::vector<string>`, según corresponda. Utilice el estándar C++14 en la solución de sus problemas. No olvide compilar con los *flags* apropiados para detectar *warnings* y errores.

1. Considere los algoritmos básicos de ordenamiento `insertion sort`, `selection sort` y `bubble sort`. Responda y justifique detalladamente las siguientes preguntas para cada uno de estos algoritmos. Sus respuestas pueden estar fundamentadas en el pseudocódigo de los mismos. Exprese estos resultados en términos del conjunto de funciones $O(f(N))$, donde $f(N)$ debe ser definida como una cota estricta (*tight bound*).

   a) ¿Cuál es la complejidad computacional del almacenamiento en memoria para el peor caso?

   b) ¿Cuál es el número de comparaciones e intercambios (*swaps*) para el mejor caso y el peor caso?

   c) ¿Cómo cambian las complejidades computacionales en $1a$ y $1b$ si todos los elementos a ser ordenados son iguales?

   d) ¿Cómo cambian los resultados de su análisis en $1a$ y $1b$ si los elementos están quasi-ordenados (*almost sorted*)?

   Definición [*Almost Sorted*]: Un arreglo está quasi-ordenado cuando sus elementos están, a lo sumo, a k posiciones de su posición final cuando el arreglo está completamente ordenado.

2. [*Bidirectional bubble sort*.] Este algoritmo, también conocido como *cocktail shaker sort*, es una variación estable de bubble sort. La idea es ordenar en ambas direcciones en cada recorrido del arreglo. De forma tal que luego de la primera iteración, tanto el mayor como el menor elemento en el arreglo están en sus posiciones finales. Esta estrategia reduce el número de comparaciones. Implemente este algoritmo y analice su complejidad computacional para el mejor caso y el peor caso.

3. [*Double selection sort*.] Este algoritmo, variante de *selection sort*, encuentra los valores mínimo y máximo en el arreglo en cada barrida. De esta forma el arreglo se ordenará desde ambos extremos. Esto reduce el número de barridas en dos eliminando el número de veces que se repite el ciclo interno de *selection sort*. No obstante, el número de comparaciones e intercambios no cambia. Implemente este algoritmo y analice su complejidad computacional para el mejor caso y el peor caso.

4. Dado un arreglo de elementos $A = \{A[i]\}$, $i = 1, \ldots, N$, programe funciones o métodos que encuentran:

   a) El ítem mínimo z tal que $z < A[i]$ para todo i.

b) El ítem máximo z tal que $z > A[i]$ para todo i.

Analice el tiempo de ejecución de estas funciones usando el modelo RAM (y el modelo de comparaciones, si aplica). ¿Cuál es el número de comparaciones máximo que ejecuta sus algoritmos?

5. Usando *selection sort*, *insertion sort* y *bubble sort*, escriba un programa que encuentra la k-ésima estadística de orden (*k-th order statistics*), dado un arreglo de enteros A de tamaño N. Proponga dos algoritmos con tiempos de ejecución:

   a) $O(N^2)$.

   b) $O(k \cdot N)$.

   Calculando el tiempo de ejecución para cada algoritmo $T(N)$ dentro del modelo RAM, demuestre que $T(N)$ pertenece a los conjuntos de funciones mencionados anteriormente.

6. Dado un arreglo $A = \{A[i]\}$, $i = 1, \ldots, N$ de números enteros, de longitud N, encuentre su media (*mean*) y mediana (*median*). La media $\overline{A}$ de un arreglo A se define como

$$\overline{A} := \frac{1}{N} \sum_{i=1}^{N} A[i]$$

   La mediana de un arreglo *ordenado* A es definido como: el elemento del medio, cuando N es impar; y como la media de los dos elementos del medio, cuando N es par.