Práctica 4 - Ordenamiento

1. Lea la implementación provista de `bsort`.

2. Implemente el algoritmo de ordenamiento por selección para arreglos (`ssort`) y listas enlazadas (`ssortl`).

3. Implemente el algoritmo de ordenamiento por inserción para arreglos (`isort`).

4. En el algoritmo burbuja (`bsort`), cambie la estructura "`while (!ordenado)`"por un `for` que repita el bloque cierta cantidad de veces fija dependiendo de la longitud del arreglo. ¿Cuál es la cantidad mínima de iteraciones necesarias para garantizar el correcto funcionamiento del algoritmo?

5. Escriba una función `int es_permutacion(int[], int[], int)` que tome dos arreglos, y la longitud de los mismos, y verifique si uno es permutación del otro, es decir, si tienen los mismos elementos la misma cantidad de veces, sin importar el orden [1].

6. Proponga versiones generales (`bsortg`, `ssortg`, `isortg`) de las funciones de ordenamiento implementadas, que trabajen con una función de comparación de tipo:

```
typedef int (*CmpFunc)(void *, void *);
```

Por ejemplo, la signatura de la función `bsortg` será:

```
void bsortg(void *base, int tam, size_t size, CmpFunc cmp);
```

Donde `base` es un puntero al inicio del arreglo, `tam` el número de elementos, `size` el tamaño de cada elemento (por ejemplo para un arreglo de enteros, será: `sizeof(int)`) y `cmp` la función de comparación.
Re-escriba el ejemplo de `main.c` para que funcione con `bsortg` en lugar de `bsort`.

7. Implemente una función de tipo `CmpFunc` que compare dos enteros de manera inversa a la usual. Utilícela para ordenar el arreglo $[1, 4, -2, 3]$ de mayor a menor.

8. Imagine que ejecuta el algoritmo de ordenamiento por selección sobre un arreglo de 10 elementos. ¿Cuántas comparaciones (llamadas a la función `CmpFunc`) realiza? Haga un cálculo en papel, y luego programe una función de comparación que lleve cuenta de la cantidad de veces que ha sido llamada. Para esto último puede ser útil la keyword `static`. Repita esto para el algoritmo de inserción.

9. Imagine que tiene una estructura `carta`, definida de la siguiente manera

```
typedef enum {
  BASTO,
  ORO,
  ESPADA,
  COPAS
} Palo;

typedef struct carta_ {
  Palo palo;
  char numero;
} carta;
```

[1] Formalmente, una permutación entre dos arreglos a, b de longitud n es una función biyectiva $f : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ tal que $a[i] = b[f(i)]$, y el ejercicio se reduce a determinar si existe tal función permutación.

Implemente una función `CmpFunc` que dadas dos cartas, las ordene por su número, sin importar el palo.

10. Implemente `msortl` que permita aplicar el algoritmo de ordenamiento por fusión a listas enlazadas (*merge sort*). Para esto, inicialmente implemente las funciones auxiliares:

- `int longitud(slist l)`: que devuelva la longitud de la lista.
- `void particionar(slist l, slist* res1, slist* res2)`: que parta la lista dada a la mitad y devuelva las dos sublistas (guardadas en `res1` y `res2`).
- `slist fusionar(slist l1, slist l2)`: que fusione dos listas ordenadas en una sola.

11. Implemente `msort` siguiendo el algoritmo de ordenamiento por fusión para arreglos.

12. En el ejercicio anterior, puede resultar que hayan cartas distintas que tienen el mismo número, y por ende no hay a una preferencia sobre cuál debe ir antes.
Por ejemplo, al ordenar el arreglo de cartas: $[4b, 3c, 4c, 1o, 6o, 2b]$, no hay una preferencia para poner a $4b$ antes que $4c$, o viceversa.
Un algoritmo de ordenamiento se dice *estable* si mantiene el orden original de los elementos en caso de que no hubiera preferencia según la función de ordenamiento. Determinar cuáles de los algoritmos implementados son estables.

13. Implemente `qsort`, siguiendo lo visto en clase. Determine si el algoritmo es estable.

14. Modifique `qsort` con las siguientes variantes de elección de pivot:

a De pivot se elige aleatoriamente un elemento del arreglo.

b De pivot se elige el último elemento del arreglo.

c De pivot se elige el elemento medio del arreglo.

d De pivot se elige la mediana entre el primer elemento, el medio y el último.

Agregue a la función `qsort` un parámetro que permita decidir cuál elección de pivot utilizar (de tipo enum).

15. Modifique `qsort` de manera de llevar registro de la cantidad de llamadas a la función (ya sea desde otras funciones, como llamadas generadas por la recusión). Analizar la cantidad de llamadas entre las diferentes elecciones de pivot y diferentes arreglos a ordenar.

16. Implemente `hsort` siguiendo el algoritmo de ordenamiento por heap. Nota: puede ser útil la función `heapify` vista en la práctica de Heaps.