Algoritmos y Programación II (75.41, 95.15) – Curso Buchwald

1.er parcialito – 02/11/2020

Resolvé los siguientes problemas en forma clara y legible. Podés incluir tantas funciones auxiliares como creas necesarias.

Debés resolver 3 de los siguientes ejercicios. Uno entre el 1 y 3, otro entre el 4 y 6, otro entre el 7 y 9. Los ejercicios a resolver están definidos por tu padrón. Podés revisar en esta planilla los que te corresponden.

1. Implementar una función `float min_distancia(float arr[], size_t n)` ($n \geq 2$) y devuelva en $\mathcal{O}(n \log n)$ cuál es la distancia minima entre todos los elementos del arreglo `arr`. Por ejemplo, la distancia mínima en el arreglo `[1.0, 7.5, 3.4, 9.2, 13.1, 6.9]` es `0.6` (la distancia entre `7.5` y `6.9`). Indicar y justificar adecuadamente la complejidad de la función implementada.

2. Implementar una función `float suma_total(float arr[], size_t n)` que, **por división y conquista**, devuelva la suma de todos los elementos. Indicar y justificar adecuadamente la complejidad de la función implementada.

3. Implementar una función que reciba un arreglo *desordenado* de valores enteros (y su largo) que, **por división y conquista**, determine si se trata de un arreglo mágico. Un arreglo es mágico si existe un valor `i` tal que `arr[i] = i`. Por ejemplo, el arreglo `[7, 3, 8, 4, 5, 2, 9]` no es mágico, pero el arreglo `[7, 3, 8, 4, 2, 5, 9]` sí lo es, puesto que `arr[5] = 5`. Indicar y justificar adecuadamente la complejidad de la función implementada.

4. Escribir una **función** `bool pila_magica(pila_t* pila)`, que reciba una pila de punteros a enteros y devuelva `true` si la pila es mágica, `false` en caso contrario. Se considera que una pila es mágica si existe un `i` tal que el i-ésimo elemento *apilado* vale `i`. Ejemplo: `pila = [4, 8, 3, 1, 7, 6]` (tope en 6) es mágico pues el tercer elemento apilado vale 3. La pila puede ser modificada durante la ejecución de la función, pero al finalizar la misma debe quedar en el mismo estado que el original. Indicar y justificar la complejidad de la función implementada.

5. Escribir una **función** `bool cola_magica(cola_t* cola)`, que reciba una cola de punteros a enteros y devuelva `true` si la cola es mágica, `false` en caso contrario. Se considera que una cola es mágica si existe un `i` tal que el i-ésimo elemento *encolado* vale `i`. Ejemplo: `cola = [4, 8, 3, 1, 7, 6]` (primero en 4) es mágico pues el tercer elemento encolado vale 3. La cola puede ser modificada durante la ejecución de la función, pero al finalizar la misma debe quedar en el mismo estado que el original. Indicar y justificar la complejidad de la función implementada.

6. Implementar una **función** `bool ingresados_mismo_orden(cola_t* cola, pila_t* pila)` que reciba una cola y una pila y determine si sus elementos fueron **ingresados** (encolados y apilados, respectivamente) en el **mismo orden**. Considerar que los elementos de ambas estructuras podrían ser diferentes, o tener diferente cantidad (en cuyos casos debería devolverse `false`). Indicar y justificar la complejidad de la función implementada. Considerar que dos punteros guardados son iguales si apuntan a la misma dirección de memoria.

7. Implementar una **primitiva** `void pila_invertir(pila_t* pila)` para una *Pila Enlazada*, que invierta el orden de sus elementos, sin utilizar estructuras auxiliares (i.e.en espacio constante). Indicar y justificar la complejidad de la primitiva. La estructura interna de la pila es la siguiente:

```c
typedef struct pila {
    nodo_pila_t* tope;
} pila_t;
```

```c
typedef struct nodo_pila {
    void* dato;
    struct nodo_pila* anterior;
} nodo_pila_t;
```

8. Implementar una **primitiva** `void cola_invertir(cola_t* cola)` para una *Cola Dinámica/Circular*, que invierta el orden de sus elementos, sin utilizar estructuras auxiliares (i.e.en espacio constante). Indicar y justificar la complejidad de la primitiva. La estructura interna de la cola es la siguiente:

```c
typedef struct cola {
    void** datos;
    size_t capacidad, cantidad;
    size_t inicio, fin;
} cola_t;
```

9. Dada una lista enlazada definida con las siguientes estructuras:

```c
typedef struct nodo_lista {
    struct nodo_lista* prox;
    void* dato;
} nodo_lista_t;
```

```c
typedef struct lista {
    nodo_lista_t* prim;
} lista_t;
```

   Implementar en C la **primitiva** `bool lista_any(const lista_t* lista, bool (*f)(void*))` que devuelve `true` sólo si la función `f` pasada por parámetro devuelve `true` para alguno de los elementos de la lista. Indicar y justificar el orden de la función implementada.