Final

Ivan Litteri - 106223

26-03-2021

Índice

1. Ejercicio 1

1.1. Resolución

```python
def _buscar_cercano_k(a, k, inicio, fin):
    if inicio == fin:
        return a[inicio]

    medio = (inicio + fin) // 2

    # k se encuentra en el arreglo.
    if a[medio] == k:
        return a[medio]

    # k no esta en el arreglo, y su mas cercano esta en el medio.
    elif a[medio] < k and a[medio+1] > k:
        return a[medio] if k-a[medio] < a[medio+1]-k else a[medio+1]

    return buscar_k(a, k, inicio, medio-1) if k < a[medio] else buscar_k(a, k, medio + 1,
    fin)

def buscar_cercano_k(a: list, k: int, largo: int):
    if k > a[largo-1]:
        return a[largo-1] # O(1)
    elif k < a[0]:
        return a[0] # O(1)

    return _buscar_cercano_k(a, k, 0, largo, {}) # O(log n)

''' Complejidad
    Si k es mayor a todos los del arreglo o menor a todos los del arreglo,
entonces la complejidad es O(1) (mejor de los casos).
    En el peor de los casos, k esta, o no esta pero su mas cercano esta en el
medio del arreglo. Entonces en ese caso su complejidad por Teorema Maestro es:

A = 1, porque se realiza un llamado recursivo.
B = 2, porque se divide al arreglo original en 2.
C = 0, porque todas las operaciones en la recursividad son constantes.

en donde, log_{B}(A) = C => la complejidad es O(n^{C} * log(n)) = O(log n).
'''
```

2. Ejercicio 2

2.1. Resolución

a) La complejidad es $\Theta(n^2)$ porque vemos un elemento y los comparamos con todos los demás, este recorrido en lista enlazada tiene esta complejidad.

b) Utilizaría una estructura `hash`, hacer las comparaciones en rango (como en el punto a), costaría menos, ya que, si bien se recorre el diccionario, no tengo que ver los números repetidos. Entonces la complejidad es $< \Theta(n^2)$. Para lograr esto, en vez de guardar las cosas en una lista las guardo en el hash.

c) Si k crece mucho, entonces usaría una búsqueda binaria, desde la mitad de k.

d)

3. Ejercicio 3

3.1. Resolución

```python
def obtener_aristas(grafo: Grafo, vertices: set) -> list:
    aristas = []
    visitados = set()
    for v in vertices:
        visitados.add(v)
        for w in grafo.adyacentes(v):
            if w not in visitados:
                aristas.append((v, w))
    return aristas

def cantidad_aristas(grafo: Grafo, vertices: set) -> int:
    return len(obtener_aristas(grafo, vertices)) // 2

def es_completo(grafo: Grafo, vertices: set) -> bool:
    n = len(vertices)
    return cantidad_aristas(grafo, vertices) == (n*(n-1)) // 2

def _clique_tamanio_k(grafo: Grafo, v: str, k: int, visitados: set):
    visitados.add(v)

    if len(visitados) == k:
        if es_completo(grafo, visitados):
            return True
        visitados.remove(v)
        return False

    for w in grafo.adyacentes(v):
        if w in visitados:
            continue
        if _clique_tamanio_k(grafo, w, k, visitados):
            return True

    visitados.remove(v)
    return False

def clique_tamanio_k(grafo: Grafo, k: int):
    if len(grafo.obtener_vertices()) < k:
        return False
    for v in grafo:
        if _clique_tamanio_k(grafo, v, k, set()):
            return True
    return False
```

4. Ejercicio 4

4.1. Resolución

```c
lista_t *lista_filtrar_repetido(const lista_t *original)
{
    lista_t *elementos_unicos;

    if ((elementos_unicos = lista_crear()) == NULL)
    {
        return NULL;
    }

    lista_iter_t *filtro;

    if ((filtro = lista_iter_crear(original)) == NULL)
    {
        lista_destruir(elementos_unicos);
        return NULL;
    }

    hash_t *filtrados;

    if ((filtro = hash_crear()) == NULL)
    {
        lista_destruir(elementos_unicos);
        lista_iter_destruir(filtro);
        return NULL;
    }

    while (!lista_iter_al_final(filtro))
    {
        void *actual = lista_iter_ver_actual(filtro);
        if (!hash_pertenece(filtrados, actual))
        {
            lista_insertar_ultimo(elementos_unicos, lista_iter_ver_actual(filtro));
        }
        hash_guardar(filtrados, actual, NULL);
        lista_iter_avanzar(filtro);
    }

    lista_iter_destruir(filtro);
    hash_destruir(filtrados);

    return elementos_unicos;
}

/* Complejidad
    Crear las estructuras es O(1). Iterar la lista original es O(n) siendo n la
cantidad de elementos en la lista original, y las operaciones que se realizan en
cada iteracion del recorrido son O(1) (hash_pertenece, hash_guardar, y
lista_insertar_ultimo son O(1)). => la complejidad de esta primitiva es O(n).
*/
```

5. Ejercicio 5

5.1. Resolución

```python
def es_diccionario_padres_bfs(d, origen):
    origen_es_padre = False
    for v in d:
        # Si v es el origen,
        if v == origen:
            # y v no es None.
            if d[v] != None:
                return False
            # y es None.
            continue
        # Si un padre no es None y no se encuentra entre los vertices.
        if d[v] != None and d[v] not in d:
            return False
        # Si hay algun vertice distinto del origen en None.
        if d[v] == None and v != origen:
            return False
        # Si el origen es padre de al menos uno.
        if d[v] == origen:
            origen_es_padre = True

    return origen_es_padre

''' Complejidad
    La complejidad de esta funcion es O(V) ya que solo itero el diccionario de
vertices que me dan, las operaciones internas son constantes.
'''
```