Algoritmos y Estructuras de Datos 2

Final 2021-03-04

Ejercicio 1

a)

- **Los productos comprados son válidos**: Todos los productos de todas las compras son claves definidas del diccionario `productos1` (los productos comprados se identifican por código). La vuelta no es necesaria porque es válido tener productos definidos que jamas fueron vendidos.
- **Las cantidades compradas son válidas**: Todas las cantidades de todas las compras deben ser mayor o igual a 1.
- **La cantidad total de ventas por producto se corresponde con las ventas registradas**: La cantidad total de ventas de un producto en particular es la suma de la cantidad vendida de ese producto entre todas las compras de todas las personas. A su vez debe valer la vuelta, todos los productos vendidos deben estar definidos en el diccionario `ventas_por_prod`.
- **El mapeo entre el código del producto y su nombre es biyectivo**: El diccionario `productos1` debe ser el inverso de `productos2` (las claves pasan a ser significados). Necesitamos que todos los códigos mapeen a su nombre y que todos los nombres mapeen a su código.

```
Rep: estr → bool
Rep(e) ≡ true ⟺ (
    ComprasVálidasYContabilizadas(e)
    ∧ VentasPorProductoVálidas(e)
    ∧ e.productos1 =obs InvertirDicc(e.productos2)
)

ComprasVálidasYContabilizadas: estr → bool
ComprasVálidasYContabilizadas(e) ≡ true ⟺ (
    (∀p: persona)(p ∈ claves(e.compras) ⇒ (
        (∀c: compra)(c ∈ obtener(p, e.compras) ⇒ (
            c.cant ≥ 1
            ∧ def?(c.cod_prod, e.productos1)
            ∧ def?(c.cod_prod, e.ventas_por_prod)
        ))
    ))
)

VentasPorProductoVálidas: estr → bool
VentasPorProductoVálidas(e) ≡ true ⟺ (
    (∀cp: cod_prod)(cp ∈ claves(e.ventas_por_prod) ⇒ (
        obtener(cp, e.ventas_por_prod) = ContarVentas(cp, e.compras)
    ))
)

ContarVentas: cod_prod × dicc(persona → conj(compra)) → nat
Sea p = dameUno(claves(d))
Sea cs = obtener(p, d)
ContarVentas(cp, d) ≡
    if vacío?(claves(d)) then
        0
    else
        ContarVentasAux(cp, cs) + ContarVentas(cp, borrar(p, d))
    fi

ContarVentasAux: cod_prod × conj(compra) → nat
```

```
Sea c = dameUno(cs)
ContarVentasAux(cp, cs) ≡
    if vacío?(cs) then
        0
    else
        if c.cod_prod = cp then c.cant else 0 fi + ContarVentasAux(cp, sinUno(cs))
    fi

InvertirDicc: dicc(K → V) → dicc(V → K)
Sea k = dameUno(claves(d))
Sea v = obtener(k, d)
InvertirDicc(d) ≡
    if vacío?(claves(d)) then
        vacío
    else
        definir(v, k, InvertirDicc(borrar(k, d)))
    fi
```

b)

El diccionario `productos1` mapea el código de producto a su nombre. Como el enunciado dice que la cantidad de productos no está acotada, podemos asumir que consecuentemente los códigos tampoco están acotados. Una primer opción podría ser representar este diccionario con un AVL donde las operaciones resultan $O(log(n))$ donde $n = |\text{productos1}|$.

Si logramos encontrar una cota m para los códigos de producto tal que un porcentaje alto de ellos tienen un código $\leq m$ se podría representar el diccionario con 2 estructuras: un HashTable para los códigos $\leq m$ que permitiría realizar las operaciones clásicas en $O(1)$, y caso contrario el resto de los códigos se colocan en el AVL donde las operaciones resultan $O(log(n))$ donde n es la cantidad de productos que tienen un código $> m$.

El diccionario `productos2` mapea el nombre del producto a su código. En este caso podríamos representar el diccionario con un Trie ya que esta estructura es eficiente para operar con claves de tipo string. En el peor caso, las operaciones tendrían una complejidad de $O(log(k))$ donde k es el nombre de producto que estamos buscando, insertando o borrando.

Ejercicio 2

operacion es tupla ⟨ diaCompra: nat, diaVenta: nat ⟩

OptimizarOperación(**in** s: arreglo(nat)) → **out** res: operacion

res ← OptimizarOperaciónAux(s, 1, tam(s))

Complejidad: $O(nlog(n))$

OptimizarOperaciónAux(**in** s: arreglo(nat), **in** low: nat, **in** high: nat) → **out** res: operacion

```
if high - low = 0 then
    res ← ⟨ diaCompra: low, diaVenta: low ⟩
else
    mid ← (low + high) / 2
    opIzq ← OptimizarOperaciónAux(s, low, mid)
    opDer ← OptimizarOperaciónAux(s, mid + 1, high)
    opMid ← ⟨ diaCompra: BuscarMinPrecio(s, low, mid), diaVenta: BuscarMaxPrecio(s, mid+1, high) ⟩
    if Ganancia(s, opIzq) > Ganancia(s, opMid) ∧ Ganancia(s, opIzq) > Ganancia(s, opDer) then
        res ← opIzq
    else if Ganancia(s, opMid) > Ganancia(s, opDer) then
        res ← opMid
    else
        res ← opDer
    end if
end if
```

Complejidad: $O(nlog(n))$

Ganancia(in s: arreglo(nat), **in** o: operacion) $\rightarrow$ **out** res: nat

```
1: res ← s[o.diaVenta] - s[o.diaCompra]
```

Complejidad: $O(1)$

Ejercicio 3

La notación $O - \Omega - \Theta$ permite caracterizar la complejidad algorítmica asintótica en función del tamaño de la entrada. Esto es muy conveniente para comparar distintos algoritmos de forma justa en el contexto de una máquina "ideal", es decir, descartamos el impacto de performance de la máquina real en donde corren los algoritmos. A su vez, analizamos su performance en el límite cuando la entrada tiende hacia el infinito. No nos interesa caracterizar los algoritmos de forma general usando entradas pequeñas, ya que en esos casos el impacto de un algoritmo ineficiente es muchísimo menor, más aún con la performance de las computadoras modernas.

Sea $f = n^2 + 2n + 8$. Para caracterizar la complejidad asintótica solo miramos el término de mayor grado.

La notación O ("Big O") indica una cota superior para el comportamiento asintótico de una función. Es decir, la tasa de crecimiento de la función es igual o menor que la de la cota. Por ejemplo, podemos afirmar que f es $O(n^2)$, como así también generalizarlo a que f es $O(n^c)$ donde $c \geq 2$.

La notación Ω indica una cota inferior para el comportamiento asintótico de una función. Es decir, la tasa de crecimiento de la función es igual o mayor que la de la cota. Por ejemplo, podemos afirmar que f es $\Omega(n^2)$, como así también generalizarlo a que f es $\Omega(n^c)$ donde $c \leq 2$.

La notación Θ indica una cota "exacta" para el comportamiento asintótico de una función. Es decir, la tasa de crecimiento de la función está acotada por arriba y por abajo, $\pm$ una constante (pueden ser distintas). Sea $g(n)$ una función, si probamos que f es $O(g(n))$ y $\Omega(g(n))$ entonces probamos que f es $\Theta(g(n))$. En el ejemplo planteado, como f es $O(n^2)$ y $\Omega(n^2)$, podemos concluir que f es $\Theta(n^2)$.

Muchos algoritmos se comportan distinto según la entrada (no solo su tamaño, sino alguna otra característica). Simplemente por cómo funciona el algoritmo, a veces sucede que es más eficiente para cierto tipo de entradas. Es por esto que usualmente usamos estas notaciones aclarando si la entrada es el "peor caso", "mejor caso" o "caso promedio", los cuales se definen a partir de las particularidades de cada algoritmo.

Ejemplos:

- InsertionSort tiene complejidad $O(n^2)$ en el peor caso, el cual sucede cuando la entrada esta ordenada en reverso. Sin embargo, en el mejor caso, cuando la entrada ya está ordenada, la complejidad resulta $\Theta(n)$ pues el ciclo interno no realiza ningún trabajo y en efecto solo recorremos el arreglo de izquierda a derecha una sola vez con el ciclo externo. De forma general debemos caracterizar a este algoritmo como $O(n^2)$.
- SelectionSort tiene complejidad $O(n^2)$ en el peor caso, pero también es $\Omega(n^2)$ en el mejor caso (por más que el elemento ya está en su lugar correcto, el algoritmo recorre hasta el final para ver si hay algún otro elemento menor), y por lo tanto concluimos que tiene complejidad $\Theta(n^2)$ en el caso promedio.

Ejercicio 4

La algomeración primaria sucede cuando hay un conjunto de posiciones ocupadas que, cuando hay una colisión con cualquiera de ellas, el barrido debe recorrerlas a todas hasta encontrar la próxima posición libre, y así se incrementa el tamaño de la algomeración en 1.

La aglomeración primaria es un conjunto de posiciones ocupadas que degradan la performance del HashTable. Por la función de hash y la forma de realizar el barrio, sucede que si una clave hashea a cualquiera de las posiciones dentro de la aglomeración primaria

donde todas generan una colisión

Ejercicio 5

- Los axiomas del TAD Empresa están axiomatizados sobre sus propios generadores pero también sobre el generador del TAD Empleado. Esto rompe el encapsulamiento, lo correcto sería utilizar una instancia genérica `e: Empleado` en los axiomas del TAD Empresa.
- Faltan los observadores `edad` y `legajo` en el TAD Empleado. Sin ellos, es imposible de axiomatizar correctamente las operaciones `suma_edades` y `legajo` del TAD Empresa.
- Al `crear` una empresa no se indica el auto que tienen los empleados iniciales. El observador `que_auto_tiene?` solo va a poder devolver el auto de los empleados que llegaron después de haber creado la empresa.
- La operación `legajo` en el TAD Empresa debería tener una restricción pidiendo que el empleado pertenezca a los empleados de la empresa.
- Las operaciones `suma_edades` y `legajo` del TAD Empresa no son observadores básicos, deberían ser otras operaciones. Esta información ya se puede obtener con el observador `empleados` pues éste devuelve un conjunto de instancias del TAD Empleado.
- La axiomatización de `empleados` sobre el generador `llega_empleado` no está agregando el nuevo empleado al conjunto de empleados existentes.
- En el TAD Empresa hay una inconsistencia con el nombre de la operación `que_auto_trajo?` y `que_auto_tiene?`. Claramente son la misma operación pero se debe respetar el mismo nombre.