Análisis de Algoritmos
Tarea 1

17 de junio de 2022

Ejercicios

Considera el problema MIN: dado un arreglo A de n elementos comparables, devolver el valor del elemento más pequeño de A.

1. Diseña un algoritmo iterativo que resuelva el problema MIN en tiempo $O(n)$. Demuestra usando invariantes que es correcto. Demuestra que su complejidad es realmente $O(n)$.

   Respuesta:

   El algoritmo requerido es:

Algoritmo 1 Un algoritmo iterativo que resuelve el problema MIN

```
def min(a):
    aux = a[0]
    for i in a:
        if aux > i:
            aux = i
    return aux
```

   Ahora proponemos el siguiente invariante:

   Después de procesar la celda $a[j]$ del arreglo, la variable `aux` contiene al valor mínimo de entre los valores de las celdas $a[0]$ a $a[j]$.

   Demostración:

   Por inducción sobre el valor de j.

   Caso Base: Si $j = 0$, entonces `aux=a[0]`, y entonces claramente `aux` contiene el valor mínimo de entre los valores de las celdas `a[0]` a `a[j]=a[0]`, es decir, solo se ha procesado un elemento del arreglo y claramente ese elemento es el mínimo de entre todos los ya procesados.

   Hipótesis de inducción: Supongamos que después de procesar la celda `a[k]` del arreglo, la variable `aux` contiene al valor mínimo de entre los valores de las celdas `a[0]` a `a[k]`.

   Paso inductivo: Al procesar la celda `a[k+1]`, por la definición de nuestro algoritmo compararemos `a[k+1]` con el valor de `aux`, pero por la hipótesis de inducción, antes de procesar `a[k+1]`, se tendrá que: `aux=mín(a[0],..,a[k])`, entonces después de procesar `a[k+1]`:

`aux=mín(mín(a[0],...,a[k]),a[k+1]) = mín(a[0],...,a[k],a[k+1])`

por lo que entonces, después de procesar `a[k+1]`, la variable `aux` contiene al valor mínimo de entre los valores de las celdas `a[0]` a `a[k+1]`.

Ya que la condición que planteamos en el invariante se conserva hasta finalizar la ejecución del algoritmo, y como el algoritmo regresa el valor de la variable `aux`, entonces siempre regresamos el valor del elemento mas pequeño en el arreglo. Por lo tanto, el algoritmo es correcto. ■

La complejidad del algoritmo es $O(n)$ ya que nuestro algoritmo compara la variable auxiliar a la que en un principio asignamos el valor de la primer casilla del arreglo y la comparamos con cada uno de los n elementos del arreglo y actualizamos su valor si es necesario, es decir, el número de operaciones que realizamos depende directamente del tamaño del arreglo.

2. Diseña un algoritmo recursivo que resuelva el problema MIN en tiempo $O(n)$. Demuestra por inducción en el valor de n que es correcto. Demuestra que su complejidad es realmente $O(n)$.
Respuesta:

El algoritmo requerido es:

Algoritmo 2 Un algoritmo recursivo que resuelve el problema MIN

```
def minrec(a):
    aux = a[0]
    if len(a) == 1:
        return a[0]
    else:
        if aux < a[1]:
            a[1] = aux
        a.pop(0)
        return minrec(a)
```

Para mostrar que el algoritmo es correcto, debemos mostrar que el algoritmo regresa el elemento mínimo para cualquier arreglo de n elementos comparables.

Demostración:

Por inducción en el valor de n.
Caso Base: Si $n = 1$, entonces el arreglo tiene tamaño 1 y entonces el algoritmo regresa el único elemento del arreglo, que efectivamente es el mínimo.

Hipótesis de Inducción: Supongamos que para $n = k$, el algoritmo regresa el elemento mínimo del arreglo.

Paso Inductivo: Para $n = k + 1$

Si $k = 0$, entonces el arreglo tiene un solo elemento, y por el caso base, el algoritmo regresa el único elemento del arreglo, es decir, el elemento mínimo.

Si $k > 0$, entonces el arreglo tiene al menos 2 elementos, y se tienen dos casos:

- Si `aux<a[1]`, entonces `a[1]=aux` y quitamos el primer elemento del arreglo y mandamos a llamar recursivamente al algoritmo pero ahora con un arreglo de k elementos, pero por la hipótesis de inducción, esa llamada recursiva nos regresa el elemento mínimo del arreglo.
- Si `aux>a[1]`, quitamos el primer elemento del arreglo y mandamos a llamar recursivamente al algoritmo pero ahora con un arreglo de k elementos, pero por la hipótesis de inducción, esa llamada

recursiva nos regresa el elemento mínimo del arreglo.

Por lo que para $n = k + 1$, el algoritmo funciona. Por lo tanto el algoritmo es correcto. ■

La complejidad del algoritmo es $O(n)$ ya que el algoritmo va a mandarse a llamar recursivamente, tantas veces como el tamaño del arreglo, es decir n veces. Por lo que una vez más el número de operaciones que realizamos depende directamente del tamaño del arreglo.

3. Diseña un algoritmo que en tiempo lineal calcule la profundidad de un árbol binario T dado. La profundidad de un árbol se define como el máximo número de aristas que separan la raíz con alguna hoja del árbol. Demuestra que el algoritmo es correcto, y que su complejidad es lineal en el número de vértices y aristas de T.

Respuesta:

El algoritmo requerido es:

Algoritmo 3 Un algoritmo recursivo que resuelve el problema 3

```
def profundidad(T):
    if (esHoja(T) || esVacio(T)):
        return 0
    else
        return 1 + max(profundidad(subIzq(T)),profundidad(subDer(T)))
```

Para mostrar que el algoritmo es correcto, debemos mostrar que para cualquier árbol binario T, el algoritmo regresa la profundidad de T, es decir, el máximo número de aristas que separan la raíz con alguna hoja del árbol.

Demostración:

Por inducción estructural sobre T.

Caso Base: Si T es una hoja, entonces el algoritmo regresa 0, que en efecto es la profundidad del árbol ya que una hoja no tiene hijos, y por lo tanto no hay aristas que salgan de ella.

Si T es vacío, entonces el algoritmo regresa 0, que es correcto, ya que un árbol vacío no tiene aristas.

Hipótesis de Inducción: Supongamos que para árboles binarios X e Y, se cumple que el algoritmo en efecto regresa la profundidad de X e Y.

Paso Inductivo: Supongamos que se tiene un árbol T con raíz r y subárboles izquierdo y derecho X e Y respectivamente, entonces:

- Si X e Y son hojas, entonces el algoritmo regresa `1+max(profundidad(X),profundidad(Y))`, y como X e Y son hojas, entonces `1+max(profundidad(X),profundidad(Y))=1+max(0,0)=0`, por lo que el algoritmo regresa 1, que es en efecto la profundidad correcta del árbol, ya que el árbol es solo una raíz con dos hojas unidas a ella.
- Si X e Y son vacíos, entonces T es una hoja, y el algoritmo regresa 0, que es la profundidad correcta para una hoja.
- Si X o Y no son hojas o vacíos, entonces el algoritmo regresa `1+max(profundidad(X),profundidad(Y))`, y por la hipótesis de inducción, podemos afirmar que `profundidad(X)` y `profundidad(Y)`, nos regresarán las profundidades de los subárboles X e Y, así que el algoritmo elegirá la máxima de entre ellas y le sumará 1 dado que estamos contando a la arista que va desde la raíz a cualquier hijo, es decir, el algoritmo regresa el máximo número de aristas que separan a la raíz con alguna hoja del árbol, que es en efecto la profundidad del árbol T.

Por lo tanto el algoritmo es correcto. ■

Dado que el algoritmo en cada llamada recursiva calcula las profundidades de los subárboles izquierdo y derecho, y además se detiene cuando el árbol es vacío o llegamos a una hoja, entonces el algoritmo termina por recorrer todo el árbol de raíz a hojas mientras va calculando las profundidades, por lo tanto la complejidad del algoritmo es lineal sobre el número de vértices y aristas del árbol.

4. Diseña un algoritmo que, dado un arreglo A de n enteros, y un entero x, devuelva el índice más pequeño tal que $A[i] == x$, o el valor -1 si x no aparece en el arreglo. Propón el invariante natural de tu algoritmo, demuéstralo, y úsalo para concluir que el algoritmo es correcto. Demuestra porqué es de tiempo $O(n)$.

Respuesta:

El algoritmo requerido es:

Algoritmo 4 Un algoritmo iterativo que resuelve el problema 4

```
def indpeq(a,x):
    min = -1
    for i in range (len(a)):
        if x == a[i]:
            min = i
            break
    return min
```

Ahora proponemos el siguiente invariante:

Después de la ejecución j del ciclo, la variable `min` contiene el indicie de la primera aparición de `x` en el arreglo o en otro caso `min=-1`

Demostración:

Por inducción sobre el valor de j.

Caso Base: Si $j = 0$ tenemos:

- Si `a[j]a[0]=x`, entonces `min=0=j`, y por lo tanto `min` contiene el índice de la primera aparición de `x`.
- Si `a[j]=a[0]`$\neq$`x`, entonces `min=-1` y no actualiza su valor, es decir x aún no aparece en el arreglo.

Hipótesis de inducción: Supongamos que después de la ejecución $j = k$ la variable `min = k`, ya que $j = k$ contiene al índice con la primera aparición de x en el arreglo o en otro caso `min = -1`

Paso inductivo: para $j = k + 1$ Por hipótesis de inducción sabemos que después de la ejecución $j = k$, `min = j` o `min = -1` entonces tenemos:

- Si `min=j` entonces el algoritmo termina ya que j es el índice que contiene la primera aparición de x en el arreglo.
- Si `min=-1` entonces el ciclo del algoritmo hace la ejecución $j = k + 1$ y obtenemos que `min=j=k+1` ya que $j = k + 1$ contiene al índice con la primera aparición de x en el arreglo o `min=-1` en otro caso.

Por lo que para $j = k + 1$, el algoritmo funciona y como la condición que se plantea en el invariante se preserva hasta final dado que en todo momento la variable `min` conserva el valor del indice con la primera aparición del elemento x o si no lo encuentra conserva su valor inicial de `-1` podemos decir el algoritmo es correcto. ■

La complejidad del algoritmo es $O(n)$ ya que en el peor de los casos, el elemento que estemos buscando en el arreglo puede estar en la última posición del arreglo o puede no estar dentro, por lo que en tales casos se terminará recorriendo todo el arreglo, y por lo tanto el número de operaciones estará relacionado con el tamaño del arreglo.