08/09/2021

Repaso I1

Complejidad y Correctitud

Carlos Paredes - cparedesr@uc.cl

Cuando un algoritmo es correcto

Cuando un algoritmo es correcto

- Termina en un número finito de pasos

Cuando un algoritmo es correcto

- Termina en un número finito de pasos
- Para todo input valido, Cumple su propósito

Cuando un algoritmo es correcto

Un buen método para demostrar correctitud en un algoritmo es utilizar inducción matemática

- Esto se debe a que en teoría debemos probar el algoritmo para cualquier input valido posible.
  Y esto es precisamente lo que la inducción nos permite hacer

Recordatorio Inducción

- Sea $S(n) = \Sigma^n\ i$, Esto es, la suma de todos los naturales hasta n

- Definimos nuestro *"algoritmo"* Como una forma mecánica y eficiente de llegar al valor buscado

$$F(n) = \frac{n(n+1)}{2}$$

¿Es nuestro algoritmo correcto?

Recordatorio Inducción

- Base Inductiva (BI):

Para la base inductiva hemos de mostrar que al aplicar el algoritmo sobre el primer elemento válido, entonces el output es correcto.

$$F(1) = S(1) \Rightarrow \frac{1(2)}{1} = 1$$

- Hipotesis de Induccion (HI):

En la hipótesis inductiva hemos de declarar nuestra hipótesis, para posteriormente utilizarla de forma inductiva

$$F(n) = \frac{n(n+1)}{2} = S(n) = \sum_{i=0}^{n} i$$

Recordatorio Inducción

- Tesis Inductiva (TI):

En este paso hemos de demostrar que si F(n) se cumple, entonces F(n+1) también ha de cumplirse (F(n) → F(n+ 1))

$$F(n+1) = F(n) + (n+1) = \frac{n(n+1)}{2} + n + 1 = \frac{n^2 + n + 2n + 2}{2}$$

$$\frac{n^2 + 3n + 2}{2} = \frac{(n+2) * (n+1)}{2}$$

$$F(n+1) = F(n) + (n+1) = S(n) + (n+1)$$

Por Inducción, F(n) es Correcto

GnomeSort Correctitud

Algoritmo de Sorting muy simple que es similar a InsertionSort.
Procedimiento:

Dado un array, $A = [a_0, a_1, \ldots, a_n]$

GnomeSort Correctitud

Algoritmo de Sorting muy simple que es similar a InsertionSort.
Procedimiento:

Dado un array, $A = [a_0, a_1, \ldots, a_n]$

1. Observa el valor de la lista actual y el anterior (si no hay anterior, asigna 0)

1. Si el anterior es menor, avanza 1 casilla en la lista. Si es mayor los intercambia y retrocede una casilla

1. Si no hay más casillas adelante, la lista está ordenada. Si no, vuelve a 1.

Pseudocódigo

```
gnomeSort(a[]):
    pos := 0
    while pos < length(a):
        if (pos == 0 or a[pos] >= a[pos-1]):
            pos := pos + 1
        else: swap a[pos] and a[pos-1]
     pos := pos - 1
```

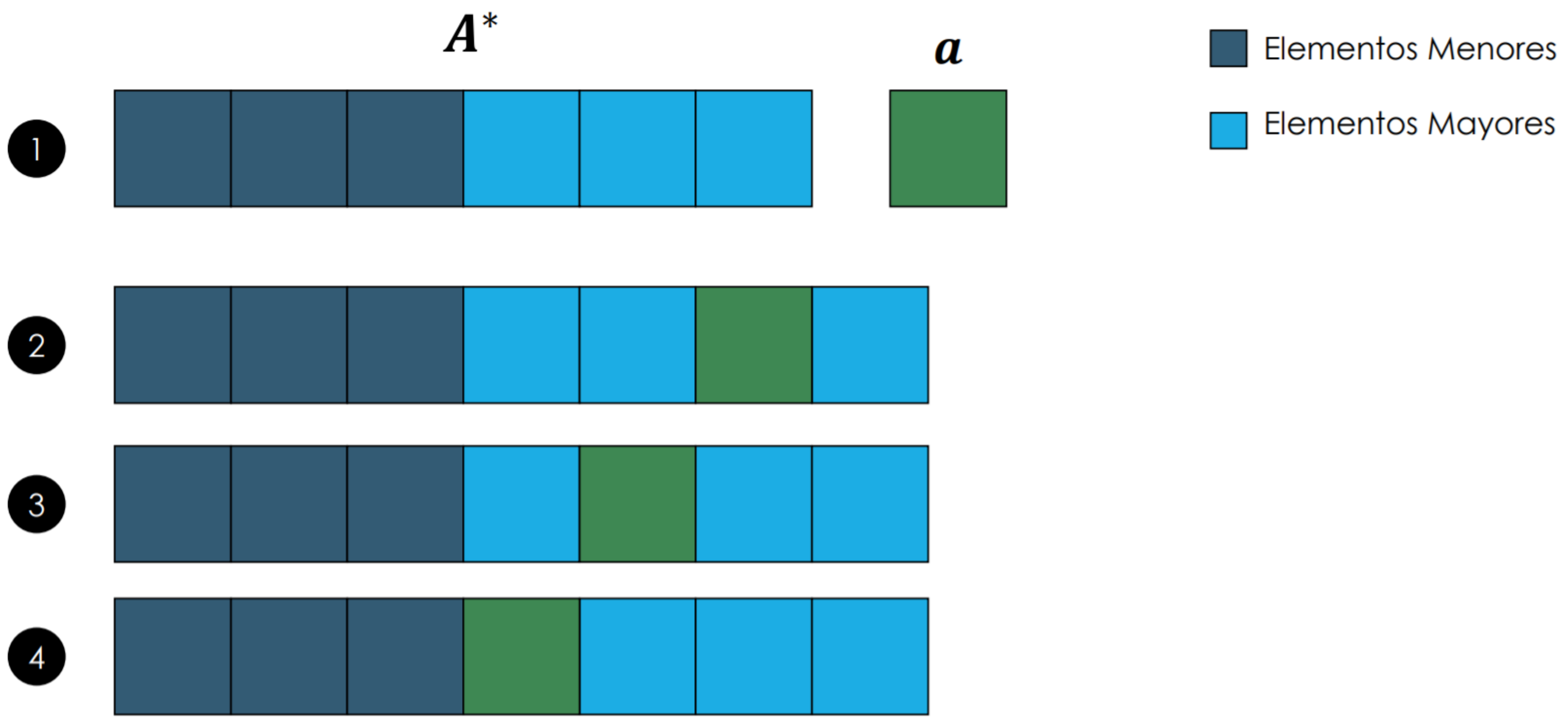
¿Es Correcto el algoritmo?

1. Demostrar que cumple con su objetivo

BI. Claramente GnomeSort($[a]$)=$[a]$ lo que esta ordenado, Dado que es un array de solo un elemento. Por ende la base inductiva se cumple

HI. GnomeSort(A) retorna una lista ordenada

TI. GnomeSort($A \cup [a]$). Por como funciona GS, esto es equivalente a $\mathrm{GS}(\mathrm{GS}(\mathrm{A}) \cup [\mathrm{a}])$.
Llamemos A^*=GS(A), la lista A ordenada.
Entonces, $GS(A* \cup [a])$. Si a es mayor a todos los elementos, entonces ya esta ordenado. Si a es menor, entonces se realizaran swaps avanzando a hasta la posición donde le corresponda


A^*
a
Elementos Menores
Elementos Mayores
1
2
3
4

¿Es Correcto el algoritmo?

2. ¿Termina en un numero de pasos finito?

Dado que cualquier array valido ha de ser finito y demostramos que el algoritmo en efecto ordena arrays validos, entonces podemos concluir que GnomeSort termina en un numero finito de pasos.

Dado que termina en numero finito de pasos y además cumple su objetivo, entonces GnomeSort es correcto

Insertion Sort - Complejidad

Una observación que hicimos en clase sobre los algoritmos de ordenación por comparación de elementos adyacentes, p.ej., *insertionSort*( ), es que su debilidad (en términos del número de operaciones que ejecutan) radica justamente en que sólo comparan e intercambian elementos adyacentes.

Así, si tuviéramos un algoritmo que usara la misma estrategia de *insertionSort*( ), pero que comparara elementos que están a una distancia > 1 entre ellos, entonces podríamos esperar un mejor desempeño

a) Calcula cuántas comparaciones entre elementos hace *insertionSort*( ) para ordenar el siguiente arreglo a de menor a mayor;
muestra que entiendes cómo funciona *insertionSort*( ): a = [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1].

Insertion Sort - Complejidad

Insertion Sort - Complejidad

insertionSort( ) coloca el segundo elemento ordenado con respecto al primero.
luego el tercero ordenado con respecto a los dos primeros (ya ordenados entre ellos).
luego el cuarto ordenado con respecto a los tres primeros (ya ordenados entre ellos), etc.

En el caso del arreglo a, *insertionSort*( ) básicamente va moviendo cada elemento, 10, 9, ..., 1, hasta la primera posición del arreglo.

Para ello, el 10 es comparado una vez (con el 11), el 9 es comparado dos veces (con el 11 y con el 10), el 8 es comparado tres veces (con el 11, el 10 y el 9), y así sucesivamente; finalmente, el 1 es comparado 10 veces.

Luego el total de comparaciones es 1 + 2 + 3 + ... + 10 = 55.

a = [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Insertion Sort - Complejidad

```
shellSort(a):
   gaps[] = {5,3,1}
   t = 0
   while t < 3:
      gap = gaps[t]
      j = gap
      while j < a.length]:
         tmp = a[j]
         k = j
         while k >= gap and tmp < a[k-gap]:
            a[k] = a[k-gap]
            k = k-gap
         a[k] = tmp
         j = j+1
   t = t+1
```

b) Calcula ahora cuántas comparaciones entre elementos hace el siguiente algoritmo *shellSort( )* para ordenar el mismo arreglo a.

Muestra que entiendes cómo funciona *shellSort( )*; en particular, ¿qué relación tiene con *insertionSort( )*?

Insertion Sort - Complejidad

Notemos que las comparaciones entre elementos de a se dan sólo en la comparación
$\boldsymbol{Tmp \ < a[k - gap];}$
el algoritmo realiza ***11*** de estas comparaciones con resultado *true* y otras ***17*** con resultado *false*; en total, ***28*.**

Primero, realiza ***insertionSort*** entre elementos que están a distancia 5 entre ellos (según las posiciones que ocupan en a, no en cuanto a sus valores): el 6 con respecto al 11, el 5 *c/r* al 10, el 4 *c/r* al 9, el 3 *c/r* al 8, el 2*c/r* al 7, el 1 *c/r* al 11, y el 1 *c/r* al 6

Luego, realiza ***insertionSort*** entre elementos que están a distancia 3 (nuevamente, según sus posiciones en el arreglo): el 2 *c/r* al 5 y el 7 *c/r* al 10.

Finalmente, realiza ***insertionSort*** entre elementos que están a distancia 1: el 3 *c/r* al 4 y el 8 *c/r* al 9; estos son los dos únicos pares de valores que aún están "*desordenados*" al finalizar el paso anterior.

Insertion Sort - Complejidad

c) Tenemos una lista de N números enteros positivos, ceros y negativos. Queremos determinar cuántos tríos de números suman 0. Da una forma de resolver este problema con complejidad mejor que $O(N^3)$.

Insertion Sort - Complejidad

c) Tenemos una lista de N números enteros positivos, ceros y negativos. Queremos determinar cuántos tríos de números suman 0. Da una forma de resolver este problema con complejidad mejor que $O(N^3)$.

Se puede hacer en tiempo $O(n^2 logn)$:
Primero, ordenamos la lista de menor a mayor, en tiempo $O(nlogn)$.
Luego, para cada par de números, sumamos los dos números y buscamos en la lista ya ordenada, empleando búsqueda binaria, un número que sea el negativo de la suma;

Si lo encontramos, entonces incrementamos el contador de los tríos que suman 0. Hay $O(n^2)$ pares (los podemos generar sistemáticamente con dos loops, uno anidado en el otro) y cada búsqueda binaria se puede hacer en tiempo $O(logn)$.