UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Algoritmos y estructuras de datos II
Análisis de complejidad por Casos

Ejercicio 1: Demuestre que $6n^3 \neq O(n^2)$.

Demostración:

T(n) es O(f(n)) si existen constantes positivas c y n0 tal que:
$T(n) \leq cf(n)\ cuando\ n \geq n0$

Para demostrar que 6n^3 no es O(n^2), necesitamos mostrar que no hay factor constante "c" y ningún tamaño de entrada "n0" tal que 6n^3 sea siempre menor o igual a cn^2 para todos los tamaños de entrada mayores o iguales a n0.

Demostraremos por contradicción, supongamos que 6n^3 es O(n^2), lo que significa que existen constantes C y n0 tales que:

$6n^3 \leq cn^2\ para\ todo\ n \geq n0$

Dividiendo ambos lados por n^2, obtenemos:

$6n \leq c\ para\ todo\ n \geq n0$

Pero esto no es cierto ya que 6n crece mucho más rápido que cualquier múltiplo constante de n para valores grandes de n. Por lo tanto, tenemos una contradicción y nuestra suposición de que 6n^3 es O(n^2) es falsa.

Por tanto, podemos concluir que 6n^3 no es O(n^2)

Ejercicio 2: ¿Cómo sería un array de números (mínimo 10 elementos) para el mejor caso de la estrategia de ordenación Quicksort(n)?

El mejor caso para QuickSort(n) es aquel array donde los elementos esten balanceados alrededor de un pivot, cuya implementación sea que el pivot sea el elemento central del array. Ese balanceo significa que a un lado del pívot están todos los elementos menores que él, y del otro lado del pívot están los elementos mayores que él. Por ejemplo:

```
array = [2,5,8,12,3,6,20,50,60,43,56,80,90,120]
```

O(n log n)

UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Ejercicio 3: ¿Cuál es el tiempo de ejecución de la estrategia Quicksort(A), Insertion-Sort(A) y Merge-Sort(A) cuando todos los elementos del array A tienen el mismo valor?

Ej: [1,1,1,1,1,1,1,1,1,1,1,1,1,1]

QuickSort(A):

El tiempo de ejecución de Quicksort en una lista donde todos los elementos son iguales es el peor caso para este algoritmo. En el peor caso, la elección del pivote resulta en particiones desbalanceadas en cada nivel de la recursión, lo que lleva a una complejidad temporal cuadrática de O(n^2)

Insertion-Sort(A):

En este algoritmo el tiempo de ejecución depende del número de inversiones en la lista de entrada. Una inversión se define como un par de elementos en la lista que están desordenados. Cuando todos los elementos en la lista son iguales, no hay inversiones. Dado que todos los elementos son iguales las comparaciones resultan siempre en igualdad y no se necesitarán cambios en la lista. Por tanto, la complejidad temporal del Insertion-Sort será del orden O(n).

Merge-Sort(A):

En Mergesort, el tiempo de ejecución en una lista donde todos los elementos son iguales es el mejor caso para este algoritmo, O(n log n).

Ejercicio 4: Implementar un algoritmo que ordene una lista de elementos de acuerdo al siguiente criterio: siempre el elemento del medio de la lista contiene antes que él en la lista la mitad de los elementos menores que él. Explique la estrategia de ordenación utilizada.

```python
# Ejercicio 4

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

def ordenarLista(lista):
    # Copio y ordeno la lista
    nuevaLista = lista.copy()
    nuevaLista.sort()
    # Determino indice y valor del pivot
    indexPivot = (len(nuevaLista) - 1) // 2
    pivot = nuevaLista[indexPivot]
    # Variable auxiliar para contabilizar elementos a dejar por debajo
    cantidadElementosMenores = pivot // 2

    i = indexPivot
    # Una vez ordenada la lista se recorre desde el comienzo hasta el pivot
    for j in range(0,indexPivot):
        # Como la lista se ordeno antes de recorrerla, una vez iteremos la mitad de elementos menores que el pivot
        # las iteraciones por encima de esa cantidad pasaran esos valores al otro lado del pivot en la lista
        cantidadElementosMenores = cantidadElementosMenores - 1
        if cantidadElementosMenores < 0:
            i = i + 1
            # Swap de valores de la lista por debajo y por encima del pivote
            nuevaLista[j], nuevaLista[i] = nuevaLista[i], nuevaLista[j]
    return nuevaLista

print(miLista)
print(ordenarLista(miLista))
```