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))
```

- **Complejidad peor caso: O(n log n)**
- **Complejidad mejor caso: O(n log n)**
- **Complejidad caso promedio: O(n log n)**

Ejercicio 5: Implementar un algoritmo Contiene-Suma(A,n) que recibe una lista de enteros A y un entero n y devuelve True si existen en A un par de elementos que sumados den n. Analice el costo computacional.

```python
def contieneSuma(A,n):
    # Creo un conjunto vacio
    S = set()
    for num in A:
        # Recorro la lista y voy revisando si existe el complemento del numero actual en esa lista
        # si no existe, lo agrego, si existe significa que hay un par de valores cuya suma me da el entero buscado
        if n - num in S:
            return True
        S.add(num)
    return False

print(contieneSuma([1,2,3],5))
```

En este caso, el costo computacional sería de O(n) ya que se recorre la lista A una sola vez y cada operación de búsqueda en el conjunto es de tiempo constante en promedio.

UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Ejercicio 6: Investigar otro algoritmo de ordenamiento como BucketSort, HeapSort o Radix Sort, brindando un ejemplo que explique su funcionamiento en un caso promedio. Mencionar su orden y explicar sus casos promedio, mejor y peor.

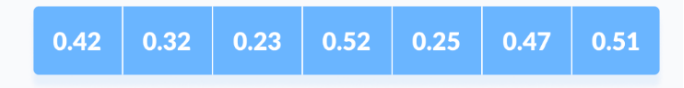
Algoritmo Bucket Sort

Bucket Sort es un algoritmo de ordenamiento que divide los elementos de un arreglo desordenado en varios grupos llamados "buckets" o "cubetas". Cada "bucket" se ordena utilizando alguno de los algoritmos de ordenamiento adecuados o aplicando recursivamente el mismo algoritmo de ordenamiento de "buckets".
Finalmente, los "buckets" ordenados se combinan para formar un arreglo final ordenado.

El enfoque de Scatter Gather (Dispersión y Recolección) se puede entender como el proceso del Bucket Sort. En este enfoque, los elementos primero se distribuyen o "dispersan" en "buckets" o "cubetas", luego los elementos en cada "bucket" se ordenan. Finalmente, los elementos se "recolectan" o se combinan en orden para formar el arreglo ordenado final.

Funcionamiento de Bucket Sort

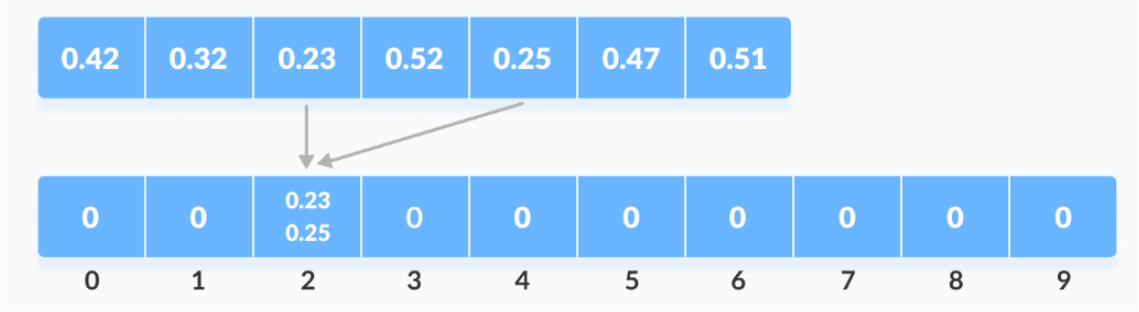
1- Suponer que el array de entrada es:



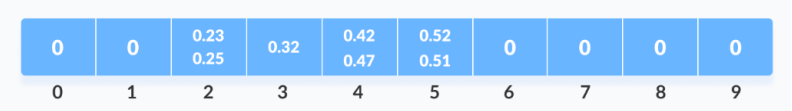
UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Creamos un array de tamaño 10. Cada posicion de este array es utilizado como un bucket para almacenar elementos.

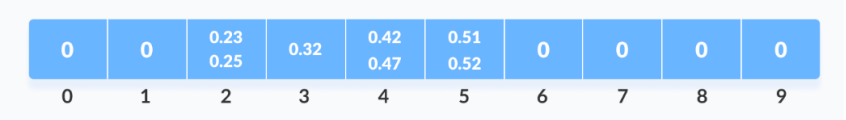
2- Insertar elementos en los "buckets" o "cubetas" del arreglo. Los elementos se insertan según el rango del "bucket". En nuestro ejemplo de código, tenemos "buckets" o "cubetas" cada uno con un rango desde 0 a 1, 1 a 2, 2 a 3, ... (n-1) a n.
Supongamos que se toma un elemento de entrada .23. Se multiplica por el tamaño = 10 (es decir, .23*10=2.3). Luego se convierte en un entero (es decir, 2.3 ≈ 2). Finalmente, .23 se inserta en el "bucket" o "cubeta" número 2.



De manera similar, .25 también se inserta en el mismo "bucket" o "cubeta". Cada vez, se toma el valor piso del número de punto flotante.
Si tomamos números enteros como entrada, tenemos que dividirlos por el intervalo (10 aquí) para obtener el valor piso. De manera similar, otros elementos se insertan en sus respectivos "buckets" o "cubetas".

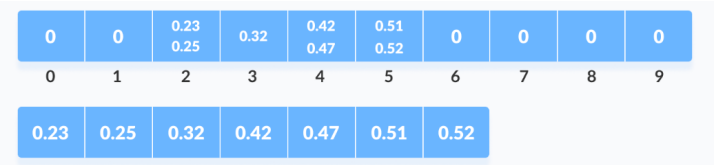


3- Los elementos de cada bucket se ordenan usando cualquier algoritmo de ordenamiento. Aqui, hemos utilizado el quicksort.



4- Los elementos de cada bucket se agrupan.

Esto se hace iterando a través del bucket e insertando un elemento individual en el array original en cada iteración. El elemento del bucket es borrado una vez es copiado en el array original.



UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Pseudocódigo

Bucket Sort Algorithm

```
bucketSort()
  create N buckets each of which can hold a range of values
  for all the buckets
    initialize each bucket with 0 values
  for all the buckets
    put elements into buckets matching the range
  for all the buckets
    sort elements in each bucket
  gather elements from each bucket
end bucketSort
```

```python
# Bucket Sort in Python


def bucketSort(array):
    bucket = []

    # Create empty buckets
    for i in range(len(array)):
        bucket.append([])

    # Insert elements into their respective buckets
    for j in array:
        index_b = int(10 * j)
        bucket[index_b].append(j)

    # Sort the elements of each bucket
    for i in range(len(array)):
        bucket[i] = sorted(bucket[i])

    # Get the sorted elements
    k = 0
    for i in range(len(array)):
        for j in range(len(bucket[i])):
            array[k] = bucket[i][j]
            k += 1
    return array
```

Bucket Sort Complexity

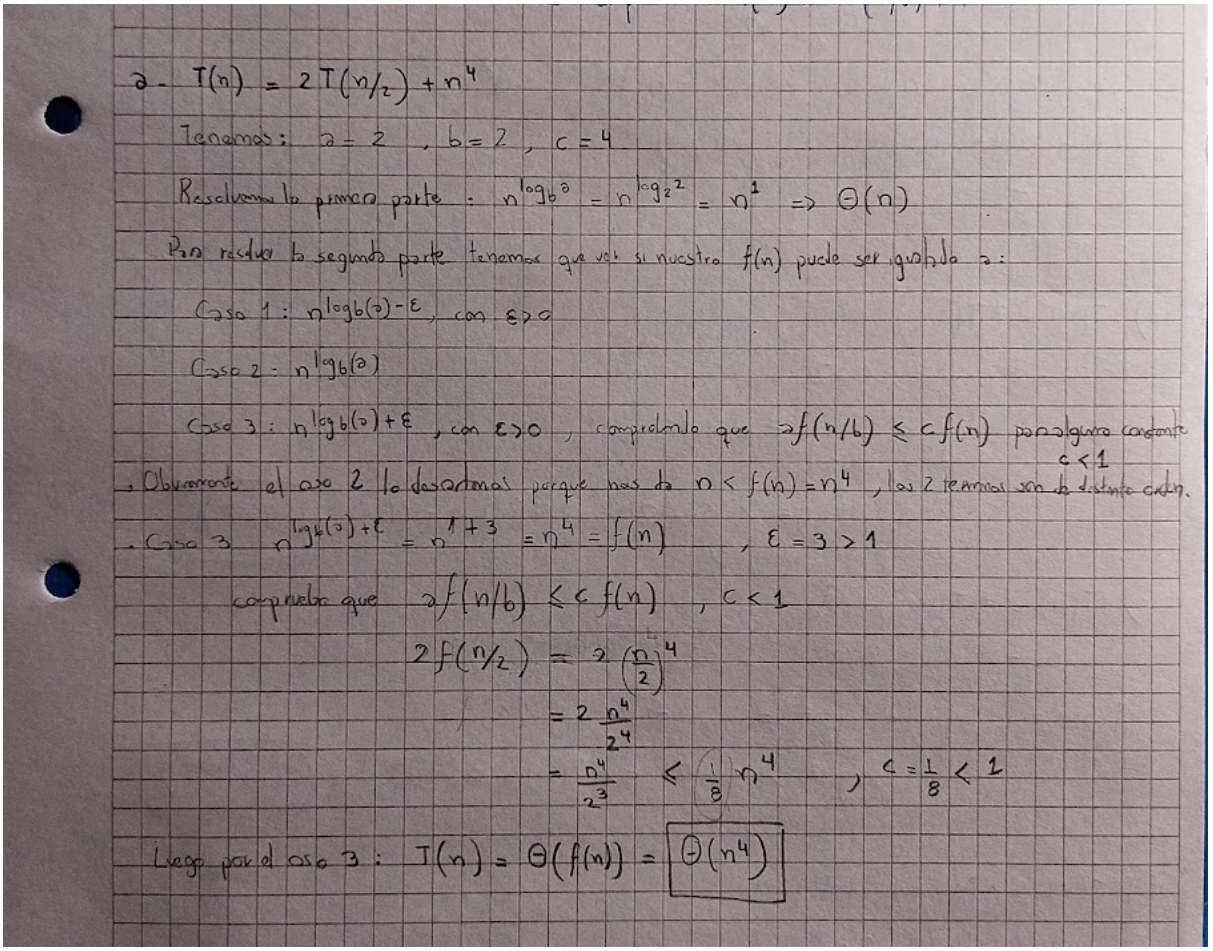
Time Complexity	
Best	O(n+k)
Worst	$O(n^2)$
Average	O(n)
Space Complexity	O(n+k)
Stability	Yes

Complejidad del caso promedio O(n):

Bucket sort es principalmente útil cuando la entrada está uniformemente distribuida en un rango. Por ejemplo, considera el siguiente problema. Ordenar un gran conjunto de números de punto flotante que están en un rango de 0.0 a 1.0 y están uniformemente distribuidos en todo el rango. ¿Cómo ordenamos los números de manera eficiente?
Una forma simple es aplicar un algoritmo de ordenamiento basado en comparaciones. El límite inferior para el algoritmo de ordenamiento basado en comparaciones (Merge Sort, Heap Sort, Quick-Sort, etc.) es Ω(n Log n), es decir, no pueden hacerlo mejor que n Log n.

La idea es utilizar Bucket Sort. Que en un caso uniformemente distribuido la complejidad de su caso promedio es O(n)

Ejercicio 7: A partir de las siguientes ecuaciones de recurrencia, encontrar la complejidad expresada en Θ(n) y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para n <= 2. Resolver 3 de ellas con el método maestro completo: T(n) = a T(n/b) + f(n) y otros 3 con el método maestro simplificado: T(n) = a T(n/b) + n^c



b) $T(n) = 2T\left(\frac{7n}{10}\right) + n$

Tenemos: $a = 2$, $b = \frac{10}{7}$, $c = 1$

Resolvemos la primera parte: $n^{\log_b(a)} = n^{\log_{\frac{10}{7}}(2)} = n^{1,94\ldots} \approx n^2$

La segunda parte, tenemos: $f(n) = n \Rightarrow n = O\left(n^{1,94-\varepsilon}\right)$, $\varepsilon > 0$

Luego $T(n) = \Theta(n^{1,94})$ (primer caso método maestro)

c) $T(n) = 16T(n/4) + n^2$

Tenemos $a = 16$, $b = 4$, $c = 2$

Resolvemos la primera parte $n^{\log_b(a)} = n^{\log_4(16)} = n^2$

$f(n) = n^2 = n^{\log_b(a)}$ (segundo caso)

Luego $T(n) = \Theta(n^2 \cdot \lg n)$

Los siguientes ejercicios se resuelven utilizando método maestro simplificado

d) $T(n) = 7T(n/3) + n^2$

$a = 7$, $b = 3$, $c = 2$

Tenemos: $\log_b a = \log_3 7 = 1,77 <$

$\log_3 7 < c = 2 \Rightarrow T(n) = \Theta(f(n)) = \boxed{\Theta(n^2)}$ (caso 3)

e) $T(n) = 7T(n/2) + n^2$

$a = 7$, $b = 2$, $c = 2$

Tenemos: $\log_b a = \log_2 7 = 2,807$

$\log_2 7 > c = 2 \Rightarrow T(n) = \Theta(n^{\log_b(a)}) = \boxed{\Theta(n^{2,807})}$ (caso 1)

f) $T(n) = 2T(n/4) + \sqrt{n}$

$a = 2$, $b = 4$, $c = \frac{1}{2}$

Tenemos: $\log_b a = \log_4(2) = \frac{1}{2} = c \Rightarrow T(n) = \Theta(n^c \lg n) = \boxed{\Theta(n^{\frac{1}{2}} \lg n)}$ (caso 2)