Análisis de Algoritmos
Unidad 3: Ordenamiento

Ing. Matias Valdenegro T.

Universidad Tecnológica Metropolitana

17 de enero de 2012

Ordenamiento

El problema consiste en ordenar un arreglo de pares llave-valor (el cual se supone desordenado) de mayor a menor (o menor a mayor).

Propiedades

- Estabilidad, un algoritmo de ordenacion estable mantiene el orden relativo de registros con llaves de igual valor.
- Uso de memoria, la complejidad espacial generalmente es $O(1)$ para algoritmos "in-place", pero existen algoritmos que usan espacio adicional al ejecutarse.

Tipos

- Ordenacion por Comparacion.
- Ordenacion sin Comparacion.

Ordenacion por Comparacion

Estos tipos de algoritmos solo usan un operador de comparacion entre elementos, y el resultado de la comparacion se usa para decidir el reordenamiento de los elementos.

- El costo de un algoritmo de ordenacion por comparacion esta siempre en $\Omega(n \log n)$.
- Esto implica que el costo minimo de ordenar al comparar llaves es $n \log n$.
- En general estos algoritmos trabajan "in-place", osea no usan memoria adicional.

Intercabio

```
void intercambio(int *a, int i, int j)
{
    int aux = a[i];
    a[i] = a[j];
    a[j] = aux;
}
```

Ordenacion sin Comparacion

Estos algoritmos no usan un operador de comparacion para ordenar el arreglo, sino que se sustentan en propiedades de los valores de las llaves, por ejemplo:

- Si las llaves son numeros en cierto rango (Bucket sort).
- Las llaves son numeros de una cierta cantidad de digitos (Radix sort).
- Las llaves estan acotadas, existe una llave mas grande que el resto (Counting sort).

Ordenacion sin Comparacion

- Como no ordenan comparando llaves, la complejidad de tiempo minima $\Omega(n \log n)$ no aplica, y generalmente son algoritmos de complejidad menor a $n \log n$ (Lineales).
- Pero pueden requerir una cantidad de espacio (memoria) al ejecutarse.
- Este es el tipico trueque entre espacio (memoria) y tiempo (tiempo de ejecucion).

Ordenacion de Burbuja

La ordenacion de burbuja (Bubble sort) es un algoritm de ordenamiento bastante simple:

1. Itera el arreglo, comparando pares de elementos, si estan en el orden incorrecto, los intercambia.
2. Ejecuta el ciclo de iteracion mientras se sigan haciendo intercambios.

Ordenacion de Burbuja

```
void bubbleSort(int *a, int n)
{
    bool cambio = false;

    do {
        for(int i = 1; i < n; i++) {
            if(a[i-1] > a[i]) {
                intercambio(a, i-1, i);

                cambio = true;
            }
        }
    } while(cambio);
}
```

Complejidad

- Peor Caso: $O(n^2)$
- Caso Promedio: $O(n^2)$
- Mejor Caso: $O(n)$

El mejor caso se presenta cuando el arreglo contiene una gran cantidad de elementos ordenados.

Ordenacion por Seleccion

El algoritmo funciona de la siguiente forma:

1. Buscar el valor minimo del arreglo.
2. Intercambiarlo con el valor en la primera posicion.
3. Repetir para el resto de la lista, empezando desde la segunda posicion.

Ordenacion por Seleccion

```
void selectSort(int *a, int n)
{
    for(int i = 0; i < n; i++) {
        int minimo = i;

        for(int j = i; j < n; j++) {
            if(a[j] < a[minimo]) {
                minimo = j;
            }
        }

        intercambiar(a, i, minimo);
    }
}
```

Complejidad

- Peor Caso: $O(n^2)$
- Caso Promedio: $O(n^2)$
- Mejor Caso: $O(n^2)$

Heapsort

Heapsort usa la estructura de datos Heap (Monticulo) para ordenar un arreglo. El algoritmo es basicamente lo siguiente:

```
void heapSort(int *a, int n)
{
    Heap h;

    for(int i = 0; i < n; i++) {
        h.insertar(a[i]);
    }

    for(int i = 0; i < n; i++) {
        a[i] = h.extraer();
    }
}
```

Complejidad de las operaciones de un Heap

Las operaciones de un heap varian en complejidad dependiendo del tipo de Heap, pero aca se presentan valores comunes:

- Insercion: $O(n \log n)$
- Extraccion del minimo: $O(n \log n)$

Complejidad

- Peor Caso: $O(n \log n)$
- Caso Promedio: $O(n \log n)$
- Mejor Caso: $O(n)$
- Complejidad espacial: $O(n)$

Quicksort

Quicksort (Ordenacion Rapida) es un algoritmo recursivo del tipo divide-y-venceras que ordena un arreglo. La estructura del algoritmo es la siguiente:

1. Elegir un elemento de la lista, llamado pivote.
2. Reordenar la lista tal que todos los elementos mayores al valor del pivote esten despues del pivote, y los valores menores al valor del pivote esten antes del pivote.
3. Recursivamente ordenar (con Quicksort) las 2 sublistas (mayores y menores al pivote).

Quicksort

```
void quicksort(int *a, int inicio, int fin)
{
    if(fin > inicio) {
        int pivote = particion(a, inicio, fin);

        quicksort(a, inicio, pivote - 1);
        quicksort(a, pivote + 1, fin);
    }
}
```

Quicksort

```
int particion(int *a, int inicio, int fin)
{
    int pivote = (inicio + fin) / 2;
    int indice = inicio;

    intercambio(a, pivote, fin);

    for(int i = inicio; i < fin; i++) {
        if(a[i] < a[pivote]) {
            intercambio(a, i, indice);
            indice++;
        }
    }

    intercambio(a, indice, fin);
    return indice;
}
```

Complejidad

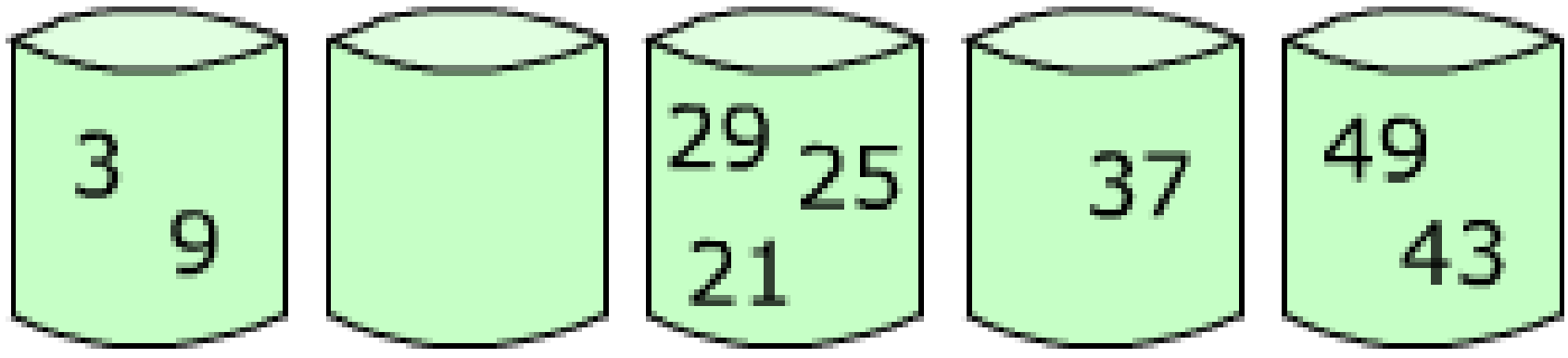
- Peor Caso: $O(n^2)$
- Caso Promedio: $O(n \log n)$
- Mejor Caso: $O(n \log n)$

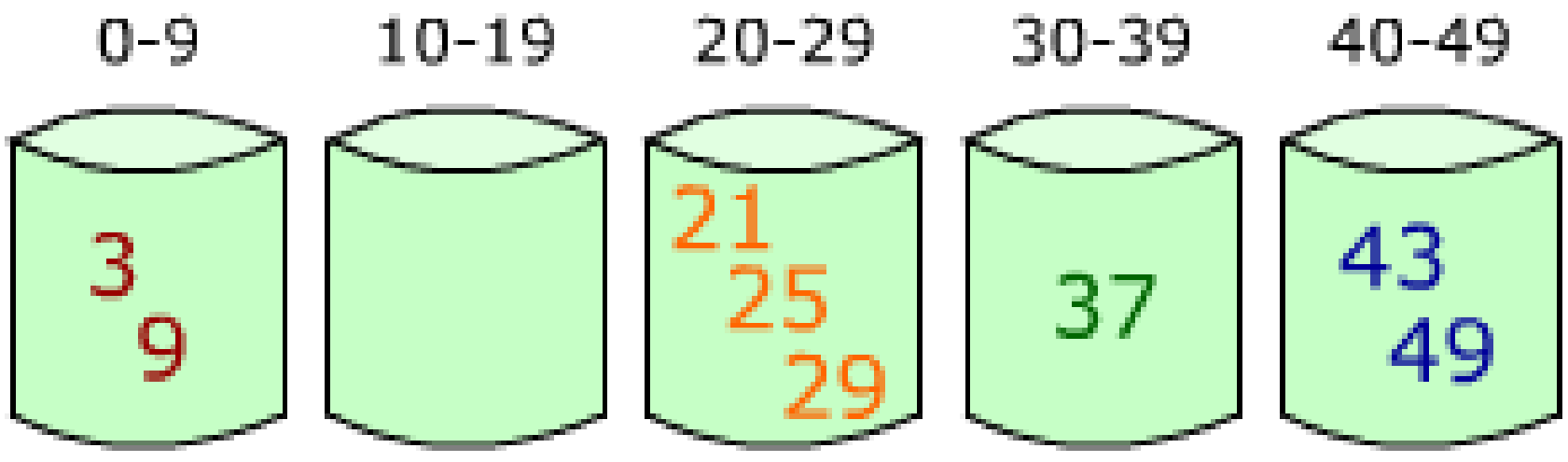
Bucket sort

Este algoritmo trabaja asumiendo que las llaves son numeros en un cierto rango, y se pueden dividir en digitos.

1. Inicializar un conjunto de K "Buckets" o contenedores.
2. Recorrer el arreglo a ordenar, insertando cada elemento en el bucket correspondiente (Se puede usar una comparacion, o insertar segun el digito de mas a la izquierda).
3. Ordenar cada bucket no-vacio usando algun otro algoritmo de ordenacion.
4. Visitar cada bucket en orden, insertando los elementos de vuelta en el arreglo original.

29 25 3 49 9 37 21 43





Complejidad

Para un arreglo de largo n y k Buckets.

- Peor Caso: $O(n^2)$
- Caso Promedio: $O(n + k)$
- Complejidad Espacial: $O(nk)$

Counting sort

Este algoritmo supone que las llaves son numeros enteros positivos. Trabaja contando la distribucion de las llaves, y usando la distribucion determina la posicion de cada llave en el arreglo ordenado.

- El supuesto es que al ordenar un arreglo a de n elementos, el valor maximo en el arreglo es k.
- Entonces se crea un arreglo c de k elementos.
- Se cuenta la distribucion de los elementos de a, usando el arreglo c.
- Una vez conocida la distribucion en c, es posible usarla para insertar los elementos en forma ordenada en a.

```
void countingSort(int *a, int *b, int n, int k)
{
    int *count = new int[k];

    for(int i = 0; i < k; i++) { count[i] = 0; }

    for(int i = 0; i < n; i++) { count[a[i]]++; }

    for(int i = 1; i < k; i++) {
        count[i] = count[i] + count[i-1];
    }

    for(int i = 0; i < n; i++) {
        b[count[a[i]] = a[i];
        count[a[j]]--;
    }
}
```

Complejidad

Pregunta

Cual es la complejidad temporal y espacial de Counting Sort?

Complejidad

Pregunta

Cual es la complejidad temporal y espacial de Counting Sort?

Respuesta

- Temporal $O(n + k)$
- Espacial $O(n + k)$

Eficiencia

Counting sort es eficiente cuando:

- El rango de valores es pequeño.
- Existen muchas llaves o valores duplicados.

Radix Sort

Idea

Radix sort esta basado en la idea de ordenar los digitos de los elementos del arreglo usando otro algoritmo de ordenacion, desde el digito menos significativo al mas significativo. Solo se requiere que el algoritmo de ordenacion sea estable.

Ejemplo

Entrada	1era Pasada	2da Pasada	3ra Pasada
329	72**0**	7**2**0	**3**29
457	35**5**	3**2**9	**3**55
657	43**6**	4**3**6	**4**36
839	45**7**	8**3**9	**4**57
436	65**7**	3**5**5	**6**57
720	32**9**	4**5**7	**7**20
355	83**9**	6**5**7	**8**39

Algoritmo

```
void radixSort(int *a, int largo, int digitos)
{
    for(int d = 1; d <= digitos; d++) {
        /* Ordena solo mirando el digito d. */
        ordenar(a, largo, d);
    }
}
```

Complejidad

Que opciones tenemos para el algoritmo de ordenacion a usar?
Supongamos que cada elemento del arreglo tiene d digitos:

Complejidad

Que opciones tenemos para el algoritmo de ordenacion a usar?
Supongamos que cada elemento del arreglo tiene d digitos:

Usando Quicksort

La complejidad de Radix Sort seria $O(dn \log n)$

Complejidad

Que opciones tenemos para el algoritmo de ordenacion a usar?
Supongamos que cada elemento del arreglo tiene d digitos:

Usando Quicksort

La complejidad de Radix Sort seria $O(dn \log n)$

Usando Bucket Sort

La complejidad de Radix Sort seria $O(d(n + k))$.
Si d es constante y $k \in \Theta(n)$, entonces Radix Sort tiene complejidad $O(n)$.

Bibliografia

Para esta unidad, se recomienda leer:

1. Capitulo 4 del Libro “Algoritmos Computacionales: Introduccion al Analisis y Diseño” de Sara Baase y Allen Van Gelder.