Ordenamiento

Nadina Martínez Carod

Tipos de Algoritmos

Fuerza Bruta

- Procesa sistemáticamente todos los posibles candidatos para la solución deun problema.
- Algoritmos simple y lentos

Divide y vencerás

- Divide y vencerás/ Disminuye y vencerás / Transforma y vencerás
- Separa el problema en problemas más pequeños para solucionarlos. Luego los combina.
- Generalmente de solución recursiva (o iterativa)
- Son más sofisticados y más rápidos

Métodos de Búsqueda

- Secuencial (Fuerza bruta)
- Binaria (Divide y vencerás)

Métodos de Ordenamiento

- Burbuja, burbuja mejorado (Fuerza Bruta)
- Selección (Fuerza Bruta)
- Inserción (Fuerza Bruta)
- Más sofisticados: (Divide y vencerás)
  - Quicksort, Mergesort, Heapsort

Método Burbuja

Idea Básica

- "Hundir" el elemento más grande al final después de comparar con los elementos adyacentes
- El elemento más pequeño "burbujea" hacia arriba

Algoritmo

- Mientras la parte desordenada tenga más de un elemento
  - Comparar todos los elementos adyacentes
  - Intercambiar los elementos que no están en orden
  - El elemento más grande siempre queda al final, se reduce la porción desordenada en 1

Método Burbuja

- **Arreglo Original**: 5, 3, 4, 9, 2
- **Primera Iteración**
  - [5, 3], 4, 9, 2 → [3, 5], 4, 9, 2
  - 3, [5, 4], 9, 2 → 3, [4, 5], 9, 2
  - 3, 4, [5, 9], 2 → 3, 4, [5, 9], 2
  - 3, 4, 5, [9, 2] → 3, 4, 5, [2, 9]
- **Segunda Iteración**
  - [3, 4], 5, 2, **9** → [3, 4], 5, 2, **9**
  - 3, [4, 5], 2, **9** → 3, [4, 5], 2, **9**
  - 3, 4, [5, 2], **9** → 3, 4, [2, 5], **9**
- **Tercera Iteración**
  - [3, 4], 2, **5, 9** → [3, 4], 2, **5, 9**
  - 3, [4, 2], **5, 9** → 3, [2, 4], **5, 9**

Eficiencia: O(n2)

- **Cuarta Iteración** :
  - [3, 2], **4, 5, 9** → [2, 3], **4, 5, 9**
- **Arreglo Final**: 2, 3, 4, 5, 9

Método Burbuja mejorado

Idea Básica

- Variante del Método de Burbuja, pero cuando se detecta que no hubo ningún intercambio en el bucle interno (es porque ya está ordenado) se corta el bucle externo

Mejoras

- Mejora la eficiencia sólo en casos en los que los elementos desordenados en una lista son pocos.
  - Ejemplo: si sólo el segundo elemento de una lista de cien elementos está fuera de lugar, la ordenación termina casi instantáneamente.
-

Método Burbuja Mejorado

- **Arreglo Original**: 5, 3, 4, 9, 7
- **Primera Iteración**
  - [5, 3], 4, 9, 7 → [3, 5], 4, 9, 7
  - 3, [5, 4], 9, 7 → 3, [4, 5], 9, 7
  - 3, 4, [5, 9], 7 → 3, 4, [5, 9], 7
  - 3, 4, 5, [9, 7] → 3, 4, 5, [7, 9]
- **Segunda Iteración**
  - [3, 4], 5, 7, **9** → [3, 4], 5, 7, **9**
  - 3, [4, 5], 7, **9** → 3, [4, 5], 7, **9**
  - 3, 4, [5, 7], **9** → 3, 4, [5, 7], **9**
- **Corta el bucle de afuera**
- **Arreglo Final: 3, 4, 5, 7, 9**

ordenado pasa a **true**

ordenado sigue **true**, entonces no necesita seguir

Antes de cada iteración, la variable **ordenado** se pone en true.

Si dentro del bucle interno se encuentran dos valores desordenados, se pone en **false**

Método Burbuja

Eficiencia

```
MODULO ordenarBurbuja (arreglo[] )   RETORNA vacío

(* a es un arreglo de enteros desordenado*)
entero: i, j
      PARA i= 0 HASTA n-1 RETORNA Ø HACER
          PARA j= 1 HASTA n-i-2 HACER
                              SI (a[j] > a [j+1])
                                  intercambio (j, j+1, arreglo)
                              FIN SI
          FIN PARA
    FIN PARA
FIN MODULO
```

O(1)

O(n)

$O(n^2)$

es de orden cuadrado

$O(n^2)$

Tiene un mejor tiempo en los mejores casos (arreglo ordenado o parcialmente ordenado),

Eficiencia: $O(n^2)$

Burbuja mejorado

```
ordenado ← falso;
i ← 0;
MIENTRAS ((i < n-1 ) AND (no ordenado)) HACER
    ordenado ← verdadero;
    PARA (j = 0 HASTA  n-i-1; j++) HACER
        SI (arreglo [j] > arreglo[j + 1]) ENTONCES
            ordenado = false;
            intercambio (j, j+1, arreglo)
        FIN SI
    FIN PARA
    i ← i + 1;
FIN MIENTRAS
```

No esto esta ordenado

```
PARA I = 0 HASTA n-1 HACER

    PARA J = 0 HASTA n-1-I  HACER

        SI (a[J] > a[J+1]) ENTONCES

            Intercambio (j, j+1, arreglo)
        FIN SI

    FIN PARA
FIN PARA
```

Burbuja

```
public static void burbujaMej(int[] arr){
 int aux, n = arr.length, i=0, j;
 boolean ordenado = false;

 while ( i< n-1 && !ordenado){
   ordenado = true;
   for( j= 0; j< n-i-1; j++){
       if(arr[j+1] < arr[j]){
           ordenado = false;
           aux= arr[j];
           arr[j]= arr[j+1];
           arr[j+1]= aux;
           }
     }
   i++;
  }
}
```

```
public static void burbuja(int[] arr){
 int aux;
 for(int i= 0; i< arr.length-1; i++){
   for(int j= 0; j< arr.length-1-i; j++){
      if(arr[j+1] <= arr[j]){
         aux= arr[j];
         arr[j]= arr[j+1];
         arr[j+1]= aux;
        }
      }
    }
  }
```

Método Selección

Idea Básica

- Encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición y así excluirlo de la lista.
- Luego el segundo mas pequeño,
- y así sucesivamente hasta ordenar todo el arreglo.

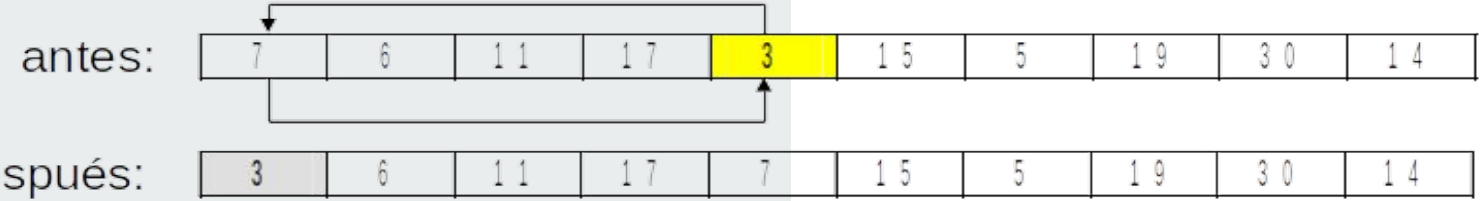
Algoritmo

- Realiza un solo intercambio en cada ejecución del ciclo externo
- Puede ser una buena elección para listas con elementos grandes y claves pequeñas.

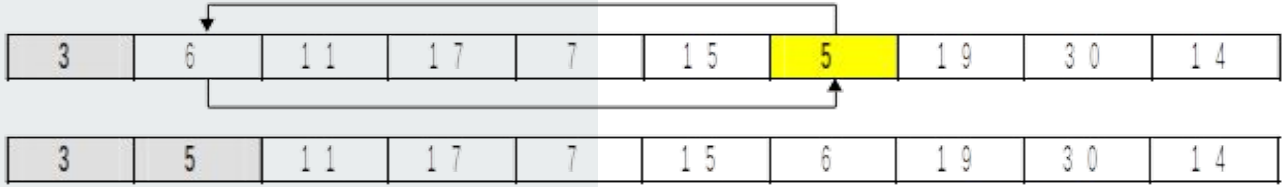
Método Selección

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
7	6	11	17	3	15	5	19	30	14

1 iteración: el valor más chico es 3, su índice es 4, intercambiar a[0] y a[4]



2 iteración: el valor más chico es 5, su índice es 6, intercambiar a[1] y a[6]



¿Cuántas iteraciones más hacen falta?

Valor más pequeño

Elementos ordenados

Método Selección

1. Buscar el número más pequeño y guardar su índice
2. Intercambiar el número más pequeño con el primer elemento del arreglo
   a. La parte ordenada del arreglo es la del principio
   b. La parte desordenada del arreglo contiene los elementos restantes
3. Repetir los pasos 1 y 2 hasta que todos los elementos estén ubicados (incrementando en 1 la longitud de la parte ordenada en cada iteración)

Ventaja: Hace pocos intercambios

Desventaja: Realiza muchas comparaciones.

```
public static void seleccion(int[] arr){
    int aux, posMenor, n = arr.length, i=0, j;

    for( i= 0; i< n; i++){
            posMenor = buscarMenor( i, arr);
            if (arr[posMenor]<arr[i]) {
                aux= arr[i];
                arr[i]= arr[posMenor];
                arr[posMenor]= aux;
                }
            }
}
```

```
public static int buscarMenor(int desde, int[] arr){
    int i, menor, posMenor;
    menor = arr[desde];
    posMenor = desde;
     for( i=desde; i< arr.length; i++){
        if (arr[i]< menor){
            menor = arr[i];
            posMenor = i;
        }
     }
  return  posMenor;
}
```

Método Inserción

Idea Básica

- "Insertar un elemento en la posición correcta dentro de una porción ordenada del arreglo
  - Es un método muy empleado para listas cortas pues no es complicado
  - Es similar a ordenar las cartas de una baraja.
  - Es el algoritmo que utilizan los jugadores de naipes
- Toma el 2do elemento del arreglo y lo intercambia o no con el 1er elemento,
- Toma el 3er elemento lo ubica donde debiera ir en forma ordenada con la 1er y 2da posición,
- realizando para esto un corrimiento de los elementos ya analizados.
- ...
-

Método Inserción

Algoritmo

1. Comienza desde el segundo elemento (primer elemento está ordenado)
2. Mientras la porción ordenada no sea el arreglo entero
3. Buscar la posición correcta para el próximo elemento, a la vez que se desplazan los elementos necesarios y se deja liberada la posición correcta
4. Insertar el elemento
5. Incrementar la porción ordenada en 1

Método Inserción

- Arreglo original: 5, 3, 4, 9, 2
- Primera iteración
  - Antes: **[5]**, 3, 4, 9, 2 Después: **[**3, **5]**, 4, 9, 2
- Segunda iteración
  - Antes: **[3, 5]**, 4, 9, 2 Después: **[3**, 4, **5]**, 9, 2
- Tercera iteración
  - Antes: **[3, 4, 5]**, 9, 2 Después: **[3, 4, 5**, 9**]**, 2
- Cuarta iteración
  - Antes: **[3, 4, 5, 9]**, 2 Después: **[**2, **3, 4, 5, 9]**
- Arreglo final: 2, 3, 4, 5, 9

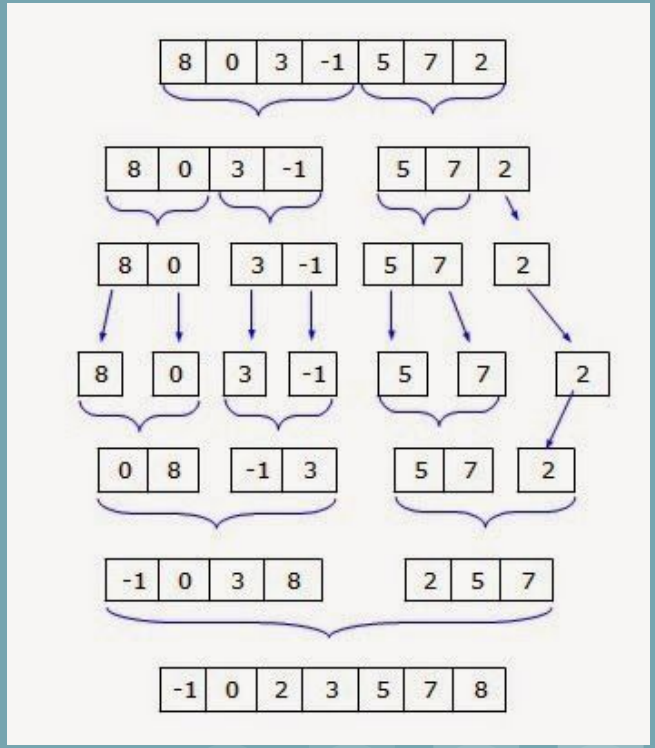
```
public static void insercionCrec(int[] arr){
        int p, temp, j, dim;

        dim = arr.length;
        for(p = 1; p < dim; p++){
            temp = arr[p];
            j = p;

            while(j>0 && temp<arr[j-1]){
                arr[j] = arr[j-1];
                j--;
            }
            arr[j] = temp;
        }
    }

}
```

Método inserción

Método Mergesort

Idea Básica

- Si la longitud de la lista es 0 ó 1, entonces ya está ordenada.
- En otro caso:
  - Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
  - Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
  - Mezclar las dos sublistas en una sola lista ordenada.



Método Mergesort

```
MODULO mergesort(ENTERO[] a,ENTERO izq,ENTERO dcho) RETORNO Ø

     ENTERO medio

     SI izq < dcho ENTONCES

        medio ← (izq + dcho)/2

        mergesort (a, izq, medio)t

        mergesort (a, medio+1, dcho)

        merge (a, izq, medio, dcho)

     FIN SI

FIN MODULO
```

Eficiencia: O(n * log n)

Método Quicksort

Idea Básica

- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
- Resituar los demás elementos de la lista a cada lado del pivote, de un lado menores, del otro mayores.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento.
- Una vez terminado este proceso todos los elementos estarán ordenados.

La eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

Método Quicksort

```
MODULO quicksort(ENTERO[] a,ENTERO i,ENTERO j) RETORNA Ø
ENTERO indice, k
    indice ← pivote(a, i, j)   //se elige el pivote
    SI indice >= 0 ENTONCES
            k ← particion (a, i, j, a[indice])
            quicksort (a, i, k-1)
            quicksort (a, k, j)
 FIN SI
FIN MODULO
```

Eficiencia: O(n * log n)

Método Heapsort

Se almacenan todos los elementos del vector en un **Arbol Heap**

```
heapsort(array A)
{
     //armar heap máximo
     armarHeapMaximo(A)

     while heap tiene elementos
        {
         //la guarda al final del arreglo
         eliminarRaiz(A)

         //empuja la raíz hacia abajo siftdown
         reacomodarHeap(A)
        }
     endwhile
}
```

Método Heapsort

Eficiencia: O(n * log n)

Cantidad de elementos n

```
armarHeapMaximo (array A)
{
    desde=1
    while desde< A.longitud()
     {
        //empuja A[inicio] hacia arriba, si su padre es menor que el
        //intercambia y sigue hasta la raiz o hasta que sea menor que su padre
        siftUp(A, desde)

        comienzo = comienzo+1
      }
    endwhile
}
```

Hace a lo sumo una visita por nivel

Heap es un arbol binario, entonces cantidad de niveles log2 n

Método Heapsort

Eficiencia: O(n * log n)

```
{
T1(n)  armarHeapMaximao(A)        O(n * log n )

       while heap tiene elementos
          {
       eliminarRaiz(A)             O(1)
T2(n)
       reacomodarHeap(A)           O(log n)
          }
       endwhile
}
```

O(n * log n)

Si *T1(n) es de O(n * log n)* *y T2(n)* es de *O(n * log n)*

Por regla de la suma: *T1(n) +T2(n)* es de *O(n * log n)*

Análisis comparativo

Algoritmo	Promedio	Mejor caso	Peor caso	Acotaciones
Burbuja mejorado	O(n*n)	O(n)	O n*n)	Se detiene al llegar al arreglo ordenado
Selección	O(n*n)	O(n*n)	O(n*n)	Aún con los datos ordenados requiere pasar por todo el arreglo
Inserción	O(n*n)	O(n)	O(n*n)	En el mejor de los casos cada inserción requiere de un tiempo constante
HeapSort	**O(n*log(n))**	**O(n*log(n))**	**O(n*log(n))**	En arreglos Requiere de espacio constante
Mergesort	O(n*log(n))	O(n*log(n))	O(n*log(n))	En arreglos requiere de mucho espacio.
Quicksort	O(n*log(n))	O(n*log(n))	O(n*n)	Depende mucho del pivote a elegir. Se debe evitar elegir como pivote el primer o último elemento