

Métodos de ordenación

Mergesort
(Ordenación por fusión)

Bases del algoritmo

- Si se tienen dos vectores ordenados, se pueden unir formando un solo gran vector ordenado. Esta operación se denomina merge y se puede implementar de forma muy eficiente.
- El algoritmo implementa una estrategia “divide y vencerás”: Tomando un arreglo de entrada, se divide en dos arreglos de mitad de tamaño, los ordena recursivamente y hace la operación merge de las dos mitades ya ordenadas.

Mergesort

Descripción del algoritmo

- Dado un arreglo de entrada $a[0..N-1]$
- Si el arreglo tiene uno ó menos elementos retornar (caso base: Un vector de ≤ 1 elemento está trivialmente en orden)
- Dividirlo en dos mitades
- Ordenar recursivamente las mitades
- Unir las mitades ya ordenadas preservando el orden (merge).

Operación merge

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    // precondition: a[lo .. mid] and a[mid+1 .. hi] are sorted subarrays
    // copy to aux[]
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }
    // merge back to a[]
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) a[k] = aux[j++];
        else if (j > hi) a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else a[k] = aux[i++];
    }
}
```

[Ver código fuente](#)

Eficiencia de merge

- Tomemos $M = hi - lo + 1$ (tamaño del intervalo)
- Accesos al arreglo
 - $2M$ en la copia a `aux[]`
 - $2M$ en las comparaciones `less()`
 - $2M$ en las asignaciones `a[..]=aux[..]`
- En total $6M$ por invocación

Eficiencia de merge

- Comparaciones entre elementos del arreglo:
 - Observamos que en el peor caso, los elementos aparecen intercalados en las dos mitades del vector.
 - En cada iteración se hace una comparación (less) y se agrega un elemento y se hace M iteraciones.
- El total de comparaciones es como máximo M .

Algoritmo Mergesort

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo) return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid + 1, hi);  
    merge(a, aux, lo, mid, hi);  
}  
  
public static void sort(Comparable[] a) {  
    Comparable[] aux = new Comparable[a.length];  
    sort(a, aux, 0, a.length-1);  
}
```

Ejemplo de ejecución

- Una invocación se descompone en un árbol binario de llamadas recursivas.

Eficiencia de Mergesort

- Modelo de costos: Comparaciones
- Proposición: El número total de comparaciones es como mínimo $\frac{1}{2} N \lg(N)$ y como máximo $N \lg(N)$.
- Demostración: Por 2 caminos
 - Descomposición en árbol
 - Por medio de una recurrencia

Análisis de la descomposición en árbol de llamadas

- Cada llamada divide el vector en 2 mitades y hace llamados recursivos sobre cada mitad.
- Se simplifica el análisis observando que un árbol de altura H tiene como máximo $2^{H+1}-1$ nodos. El árbol tiene H niveles y $H \sim \lg N$.
- Por cada nivel se tienen 2^P subarreglos, siendo P la profundidad. Cada subarreglo es de tamaño 2^{H-P} .
- Para cada nivel se hacen 2^{P-1} merge y cada merge hace 2^{H-P} comparaciones: #comparaciones por nivel: $2^{P-1} * 2^{H-P} = 2^{H-1}$.
- Finalmente son H niveles y $H \sim \lg N$, por tanto el total de comparaciones es: $\sim \lg N * 2^{\lg N - 1} \sim N \lg N$

Análisis por medio de recurrencias

- Sea $C(N)$ el número de comparaciones que hace mergesort en un arreglo de tamaño N .
- Observando que las llamadas recursivas hacen $C(N/2)$ comparaciones y el merge hace como máximo N comparaciones, se puede plantear la recurrencia:

$$C(N) = C(\lfloor N/2 \rfloor) + C(\lfloor N/2 \rfloor) + N$$

Botton-up mergesort

- La versión considerada hasta el momento es una versión top-down: Inicia desde el vector completo y lo va dividiendo en partes más pequeñas, las cuales se ordenan recursivamente.
- Es posible plantear una versión del algoritmo botton-up: Iniciar ordenando los subvectores más pequeños, fusionándolos y procediendo en niveles cada vez más grandes.

Complejidad algorítmica

- Hemos visto varios algoritmos que resuelven el problema de ordenar un arreglo:
 - Selección $\sim N^2$
 - Inserción $\sim N^2$
 - Mergesort $\sim N \lg(N)$
- El hecho de encontrar un algoritmo con un tiempo de peor caso $\sim N \lg(N)$ nos permite concluir que la complejidad de la ordenación es como máximo $\sim N \lg(N)$.

Complejidad algorítmica

- Área de la algoritmia que busca caracterizar los problemas en función de que tan eficientes son los algoritmos para resolverlos.
- El estudio de la complejidad parte de un modelo computación, *e.g.* contabilizar el número de comparaciones en un algoritmo de ordenación.
- Cabe preguntarse si existen algoritmos de ordenación que tenga un tiempo menor a $\sim N \lg(N)$ (en sentido asintótico).

Cota inferior para la complejidad algorítmica de la ordenación

- **Proposición:** Ningún algoritmo basado en comparaciones puede ordenar un vector de tamaño N con un número de comparaciones menor a $\sim \lg(N!) \sim N \lg(N)$.

Complejidad de la ordenación

Para un modelo de cómputo basado en comparaciones:

- Mergesort nos da una cota superior $\sim N \lg(N)$.
- La proposición previa nos da una cota inferior $\sim N \lg(N)$.

Se concluye que Mergesort es asintóticamente óptimo.