Ordenación

Estructuras de Datos y Algoritmos – IIC2133

Yadran Eterovic

1-2016

¿Qué es ordenación?

Proceso de reorganizar una secuencia de objetos de modo de ponerlos en algún orden lógico

- p.ej., en la lista del curso, los estudiantes están ordenados alfabéticamente
- … la cuenta de la tarjeta de crédito trae las transacciones ordenadas por fecha

¿Por qué es importante?

Es mucho más fácil buscar un dato en un arreglo ordenado:

p.ej., la antigua guía de teléfonos, las aplicaciones digitales para tocar música, los resultados producidos por los motores de búsqueda

… hace más fácil otras tareas:

p.ej., eliminar duplicados en listas de correos o de sitios Web; hacer cálculos estadísticos, tales como eliminar outliers, encontrar medianas, o calcular percentiles

… y es un subproblema importante en otras aplicaciones:

p.ej., compresión de datos, gráfica computacional, biología computacional, optimización combinatorial, y gestión de la cadena de suministro

¿Por qué estudiamos algoritmos de ordenación?

Analizarlos es una buena introducción a las técnicas de comparación de desempeño de algoritmos

Técnicas similares son eficaces para resolver otros problemas

Los usamos como punto de partida para resolver otros problemas

Son elegantes, clásicos y eficaces

Reglas

Estudiaremos algoritmos para reorganizar arreglos de ítemes, en que cada ítem contiene una clave

El objetivo de los algoritmos es reorganizar los ítemes de modo que sus claves queden ordenadas según una regla bien definida:

- la clave de cada posición no es menor que la clave en cada posición con un índice menor y no es mayor que la clave en cada posición con un índice mayor
- p.ej., orden numérico o alfabético

Propiedades de los algoritmos

Certificación:

¿Queda el arreglo siempre ordenado, independientemente del orden inicial?

Tiempo de ejecución:

Podemos demostrar hechos sobre el número de operaciones (comparaciones e intercambios) ejecutadas, para diversos modelos de datos de entrada

Memoria adicional:

Algunos algoritmos ordenan *in situ* y no usan memoria extra, excepto por un número constante de variables o un pequeño *stack* de llamadas de funciones

… otros necesitan poder guardar otra copia del arreglo que están ordenando

Tipos de datos:

Cualquier tipo de datos que defina un orden total (una relación refleja, antisimétrica y transitiva) sobre los ítemes

Dos grandes clases de algoritmos de ordenación

Algoritmos simples:

- toman tiempo O(n^2) en el peor caso,
- … y también en el caso promedio
- p.ej., *insertionSort*
- hay algoritmos simples con mejor desempeño en el caso promedio, p.ej., *shellSort* —simple de implementar, no tan simple de analizar

… y no tanto:

- toman tiempo O($n$log$n$) en el caso promedio

  p.ej., *quickSort*
- … y algunos también en el peor caso

  p.ej., *heapSort*

¿Cómo medimos el desempeño de los algoritmos de ordenación en función del número de datos, n, que queremos ordenar?

El número de pasos de un algoritmo de ordenación es proporcional a

- el número de **comparaciones** que el algoritmo hace
- el número de veces que los datos son **movidos** o **intercambiados**

¿De qué depende el desempeño de los algoritmos de ordenación simples?

Del tamaño, n, del input

… pero también

Input aleatorio:

- los tiempos de ejecución son $O(n^2)$

… y difieren por un factor constante

Input con características especiales:

- los tiempos de ejecución pueden diferir por más que un factor constante

`selectionSort`: Un algoritmo muy simple

Repetidamente, selecciona el dato más pequeño que aún no ha sido ordenado:

- primero, encuentra el dato más pequeño en el arreglo, y lo intercambia con el dato en la primera posición
- luego, encuentra el segundo dato más pequeño y lo intercambia con el dato en la segunda posición
- continúa de esta manera hasta que todo el arreglo esté ordenado

```
selectionSort(a):
    for k = 0 … n-1:
        min = k
        for j = k+1 … n:
            if a[j].key < a[min].key:
                min = j
        exchange(a[k], a[min])
```

S O R T I N G E X A M P L E
A O R T I N G E X S M P L E
A E R T I N G O X S M P L E
A E E T I N G O X S M P L R
A E E G I N T O X S M P L R
A E E G I N T O X S M P L R
A E E G I L T O X S M P N R
A E E G I L M O X S T P N R
A E E G I L M N X S T P O R
A E E G I L M N O S T P X R
A E E G I L M N O P T S X R
A E E G I L M N O P R S X T
A E E G I L M N O P R S X T
A E E G I L M N O P R S T X
A E E G I L M N O P R S T X

Sea n el número de datos a ordenar

El movimiento de datos (el intercambio, o `exchange`) queda fuera del ciclo interior:

- cada intercambio pone un dato en su posición final
- el número de intercambios es $n - 1$

El tiempo de ejecución está dominado por el número de comparaciones:

- para cada k, desde 1 hasta n, hay $n - k$ comparaciones
- el número total de comparaciones es aproximadamente $n^2/2$

El tiempo de ejecución casi no depende de cuán ordenados estén los datos:

- los números de intercambios y comparaciones son independientes de los datos de entrada

Lo único que depende del input es cuántas veces `min` es actualizado:

- en el peor caso, este número es $O(n^2)$
- en el caso promedio, este número es $O(n \log n)$

El proceso de encontrar el dato mínimo para un determinado valor de k no da información acerca de dónde estará el dato mínimo para el siguiente valor de k

En resumen,

- simple
- fácil de entender e implementar
- su tiempo de ejecución es insensible al input
- el movimiento de datos es mínimo

`insertionSort`: Un algoritmo simple

Repetidamente, considera un dato más y lo inserta en su lugar apropiado entre los datos que ya han sido considerados (y están ordenados):

- primero, toma el dato que está en la segunda posición y lo ubica correctamente con respecto al que está en la primera
- luego, toma el dato que está en la tercera posición y lo ubica correctamente con respecto a los dos primeros
- continúa de esta manera hasta que todo el arreglo esté ordenado

```
insertionSort'(a):
    for k = 1 … n:
        for j = k ∴ 1:
            if a[j].key < a[j-1].key:
                exchange(a[j-1], a[j])
```

S O R T I N G E X A M P L E
O S R T I N G E X A M P L E
O R S T I N G E X A M P L E
O R S T I N G E X A M P L E
I O R S T N G E X A M P L E
I N O R S T G E X A M P L E
G I N O R S T E X A M P L E
E G I N O R S T X A M P L E
E G I N O R S T X A M P L E
A E G I N O R S T X M P L E
A E G I M N O R S T X P L E
A E G I M N O P R S T X L E
A E G I L M N O P R S T X E
A E E G I L M N O P R S T X
A E E G I L M N O P R S T X

La versión de `insertionSort` en la próxima diapositiva mejora la anterior de tres maneras:

- detiene el `for` interior en cuanto `a[j] > a[j-1]` —el algoritmo se vuelve adaptable
- hace una primera pasada sobre el arreglo que pone el menor dato en la primera posición
- en lugar de los múltiples intercambios, va moviendo los datos más grandes una posición a la derecha

Ordenamos el arreglo `r` llamando a `insertionSort(r)`

```
insertionSort(a):
    for k = n … 1:
        if a[k].key < a[k-1].key:
            exchange(a[k-1], a[k])
    for k = 2 … n:
        j = k
        b = a[k]
        while b.key < a[j-1].key:
            a[j] = a[j-1]
            j = j-1
        a[j] = b
```

El desempeño de `insertionSort` depende principalmente del orden inicial de los datos

Sea n el número de datos a ordenar

El número de comparaciones y de movimientos es el mismo:

- aproximadamente $n^2/4$ comparaciones y $n^2/4$ movimientos en promedio
- el doble de este número, en el peor caso

Todo algoritmo (no sólo `insertionSort`) **que ordena intercambiando sólo datos adyacentes toma tiempo $\Omega(n^2)$ en promedio**

Si los datos ya están ordenados (o casi), entonces `insertionSort` es muy rápido:

- sólo hace $O(n)$ comparaciones

`shellSort` (D. Shell, 1959)

El valor de conocer las propiedades de los algoritmos más elementales

Un algoritmo rápido basado en `insertionSort`:

- `insertionSort` es lento —$O(n^2)$— para arreglos grandes desordenados porque los únicos intercambios que hace son entre datos adyacentes —los datos se mueven a través del arreglo sólo un lugar a la vez
- `shellSort` es una extensión de `insertionSort` que permite intercambios de datos que están muy separados, produciendo arreglos parcialmente ordenados que pueden ser ordenados eficientemente (usando finalmente `insertionSort`)

k-ordenación

Arreglamos el arreglo para darle la propiedad de que si tomamos cada *k*-ésimo dato (empezando en cualquier parte) tenemos una subsecuencia ordenada —un arreglo "*k*-ordenado":

- hay *k* subsecuencias ordenadas e independientes, entremezcladas en el mismo arreglo
- en la próxima diapositiva, el arreglo original (*input*) aparece primero 13-ordenado, luego 4-ordenado, y finalmente 1-ordenado (ordenado totalmente)
- al comienzo, cuando hay más desorden, las subsecuencias son cortas
- luego, cuando las secuencias son más largas, las subsecuencias están parcialmente ordenadas

input

S H E L L S O R T E X A M P L E

k = 13

P H E L L S O R T E X A M S L E

k = 4

L E E A M H L E P S O L T S X R

k = 1

A E E E H L L L M O P R S S T X

Al *k*-ordenar para valores grandes de *k*, podemos mover datos en el arreglo largas distancias

… y así hacemos más fácil *k*-ordenar para valores más pequeños de *k* :

- **¡ un arreglo *k*-ordenado que es *h*-ordenado, permanece *k*-ordenado !**

Usando este procedimiento con cualquier secuencia de valores de *k* que termine en 1, ordenaremos el arreglo

¿Cómo *k*-ordenamos (para cada valor de *k*)?

- una posibilidad es usar `insertionSort` independientemente para cada una de las *k* subsecuencias —*k* "pasadas" de `insertionSort`
- … pero como las subsecuencias son independientes, podemos hacer una sola pasada de `insertionSort`, cuidando de decrementar en *k* en vez de 1 cuando nos movemos por el arreglo

k = 13

P H E L L S O R T E X A M S L E
P H E L L S O R T E X A M S L E
P H E L L S O R T E X A M S L E

k = 4

L H E L P S O R T E X A M S L E
L H E L P S O R T E X A M S L E
L H E L P S O R T E X A M S L E
L H E L P S O R T E X A M S L E
L H E L P S O R T E X A M S L E
L E E L P H O R T S X A M S L E
L E E L P H O R T S X A M S L E
L E E A P H O L T S X R M S L E
L E E A M H O L P S X R T S L E
L E E A M H O L P S X R T S L E
L E E A M H L L P S O R T S X E
L E E A M H L E P S O L T S X R

Ordenamos el arreglo `r` llamando a `shellSort(r)`

```
shellSort(a):
    n = a.length
    k = 1
    while k < n/3: k = 3*k+1
    while k ≥ 1:
        for i = k … n-1:
            j = i
            while j ≥ k && a[j] < a[j-k]:
                exchange(a[j],a[j-k])
                j = j-k
        k = k/3
```

— 1, 4, 13, 40, 121, 364, 1093, ...

¿Qué secuencia de valores de k usamos?

Cualquier secuencia de valores sirve si el último valor es 1, pero algunas secuencias producen mejores resultados que otras

Si partimos con $k = \lfloor n/2 \rfloor$ y lo vamos reduciendo a la mitad, que es la secuencia sugerida por el propio Shell, el número de comparaciones en el peor caso es $\Theta(n^2)$

En el algoritmo anterior, sin embargo, usamos los k de la forma $(3^p - 1)/2$, partiendo del mayor valor de k que sea menor que $n/3$:

¡ el número de comparaciones en el peor caso es proporcional a $n^{3/2}$!

¿Por qué estudiamos diseño y desempeño de algoritmos?

Para lograr aumentos de eficiencia que permitan la solución de problemas que no podrían ser resueltos de otra manera

Tres algoritmos de ordenación no tan simples

`heapSort`:

- tiempo O($n$log$n$) en el peor caso —ya estudiado (¿?)

`mergeSort`:

- tiempo O($n$log$n$) en el peor caso
- necesita O(n) memoria extra

`quickSort`:

- tiempo O(n^2) en el peor caso
- tiempo O($n$log$n$) en el caso promedio
- en la práctica, parece que se desempeña mejor que `heapSort`

`mergeSort` aplica la estrategia dividir-para-conquistar

Dividir:

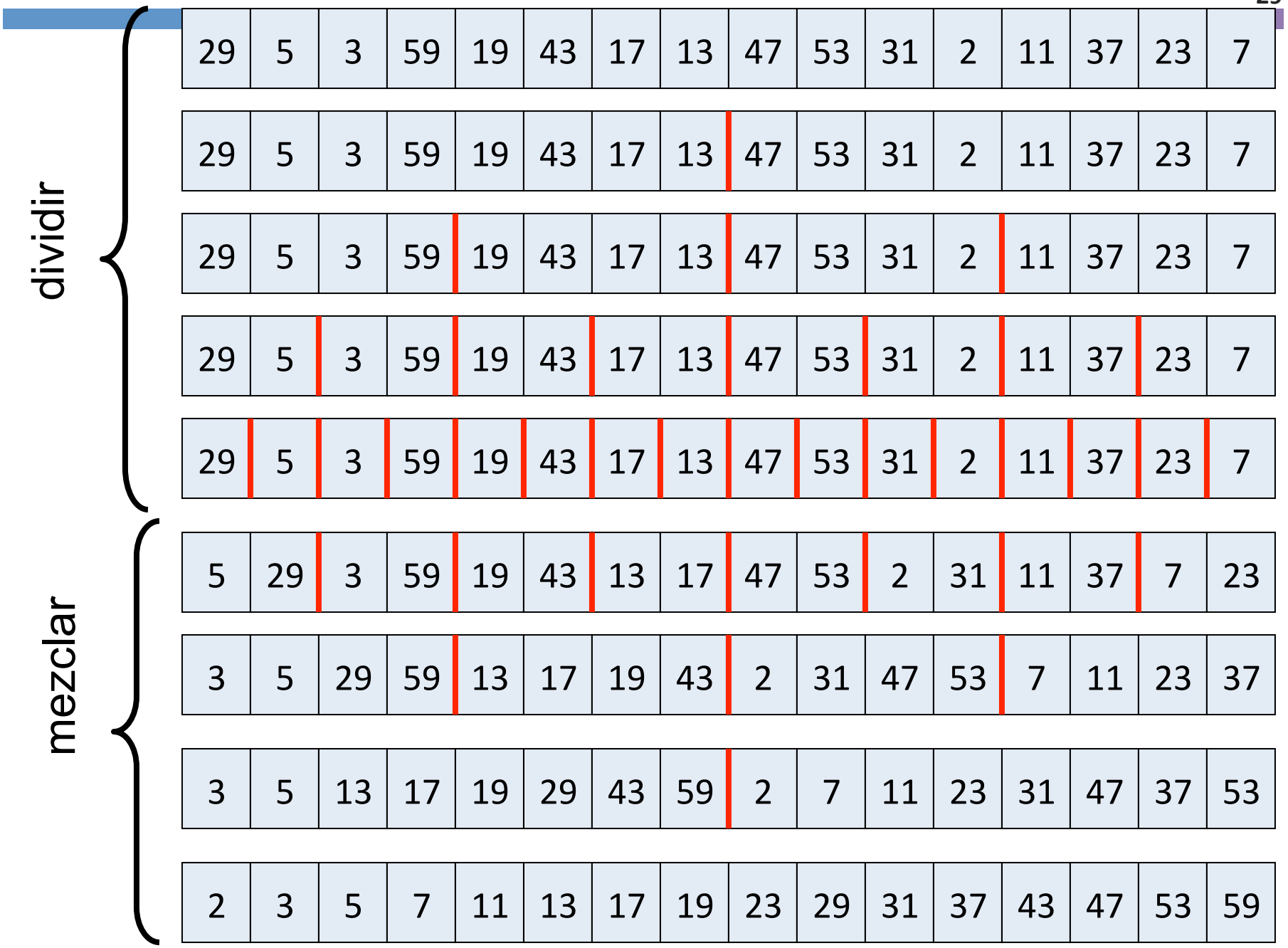
dividir la secuencia de n datos en dos subsecuencias de $n/2$ datos cada una

Conquistar:

ordenar las dos subsecuencias recursivamente usando `mergeSort`

Combinar:

mezclar las dos subsecuencias ya ordenadas, para producir la permutación ordenada de la secuencia original


dividir
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
mezclar
5 29 3 59 19 43 13 17 47 53 2 31 11 37 7 23
3 5 29 59 13 17 19 43 2 31 47 53 7 11 23 37
3 5 13 17 19 29 43 59 2 7 11 23 31 47 37 53
2 3 5 7 11 13 17 19 23 29 31 37 43 47 53 59

Ordenamos el arreglo r llamando a mergeSort(r)

```
mergeSort(a):
    tmp = a
    mergeSort(a, tmp, 0, a.length-1)

mergeSort(a, tmp, e, w):
    if e < w:
        m = (e+w)/2
        mergeSort(a, tmp, e, m)
        mergeSort(a, tmp, m+1, w)
        merge(a, tmp, e, m+1, w)
```

merge mezcla los subarreglos ordenados a[e..m-1] y a[m..w] en el subarreglo ordenado a[e..w]

```
merge(a, tmp, e, m, w):
    p = e, k = e, q = m
    while p <= m-1 && q <= w:
        if a[p].key < a[q].key:
            tmp[k] = a[p]; k = k+1; p = p+1
        else:
            tmp[k] = a[q]; k = k+1; q = q+1
    while p <= m-1:
        tmp[k] = a[p]; k = k+1; p = p+1
    while q <= w:
        tmp[k] = a[q]; k = k+1; q = q+1
    for k = e … w:
        a[k] = tmp[k]
```

El tiempo de ejecución, T(n), de `mergeSort` es O($n$log$n$), en el peor caso y también en el caso promedio

Sabemos que T(1) = 0 y T(n) = 2T(n/2) + n

Luego,

T(n)/n	=	T(n/2)/(n/2) + 1
T(n/2)/(n/2)	=	T(n/4)/(n/4) + 1
T(n/4)/(n/4)	=	T(n/8)/(n/8) + 1
...		...
T(2)/2	=	T(1)/1 + 1

Si sumamos ambos lados del signo = y cancelamos los términos que aparecen a ambos lados, obtenemos

T(n)/n	=	T(1)/1 + log$n$

Podemos reducir significativamente el tiempo de ejecución de `mergeSort()`

Usar `insertionSort()` para subarreglos pequeños, p.ej., 15 o menos elementos

Probar si el arreglo ya está ordenado, después de las llamadas recursivas a `mergeSort()` y antes de la llamada a `merge()`

Eliminar la copia al arreglo auxiliar `tmp`, intercambiando los roles de los arreglos `a` y `tmp` en cada nivel de la recursión

También podemos implementar `mergeSort()` *botom-up*