CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Ordenamiento en tiempo lineal

Objetivos

- Entender como "romper" la barrera teórica de $\Omega(n \log n)$ en tiempo para ordenamiento al utilizar más información sobre los elementos
- Tres algoritmos de ordenamiento en tiempo lineal: `Counting-Sort`, `Radix-Sort` y `Bucket-Sort`
- Analizar la correctitud y desempeño de los algoritmos

Introducción

Los algoritmos de ordenamiento basado en comparaciones son generales ya que sólo asumen:

- un **orden total** sobre los elementos en el arreglo de entrada
- un **procedimiento para comparar** dos elementos en la entrada

Bajo estas suposiciones, ordenamiento basado en comparaciones es lo único que podemos hacer y su complejidad en tiempo es $\Omega(n \log n)$

A continuación asumiremos más propiedades sobre la entrada lo que permitirá obtener algoritmos de ordenamiento más eficientes

Ordenamiento por conteo (counting sort)

Asumiremos que los elementos en la entrada son **enteros no negativos** en el rango $\{0, 1, \ldots, k\}$

En ordenamiento por conteo lo primero que se hace es contar cuantos elementos son menores a x, para cada elemento x

Una vez que la información es calculada, cada elemento x es **colocado directamente** en su posición final en el arreglo de salida

Por ejemplo, si existen 2 copias de x y existen 12 elementos menores a x en la entrada, entonces las posiciones 13 y 14 de la salida deben contener a x

Ordenamiento por conteo (counting sort)

Input: arreglo A con enteros en $\{0, \ldots, k\}$, arreglo de salida B, y k

Output: arreglo B con enteros en A ordenados de menor a mayor

```
1  Counting-Sort(array A, array B, int k)
2      let C[0...k] be new array
3      for i = 0 to k
4          C[i] = 0
5
6      for j = 1 to A.length              % frencuencia de cada elemento
7          C[A[j]] = C[A[j]] + 1
8
9      for i = 1 to k                     % cuántos elementos <= existen
10         C[i] = C[i] + C[i-1]
11
12     for j = A.length to 1      % construye arreglo de salida ordenado
13         B[C[A[j]]] = A[j]
14         C[A[j]] = C[A[j]] - 1
```


Counting-Sort en acción

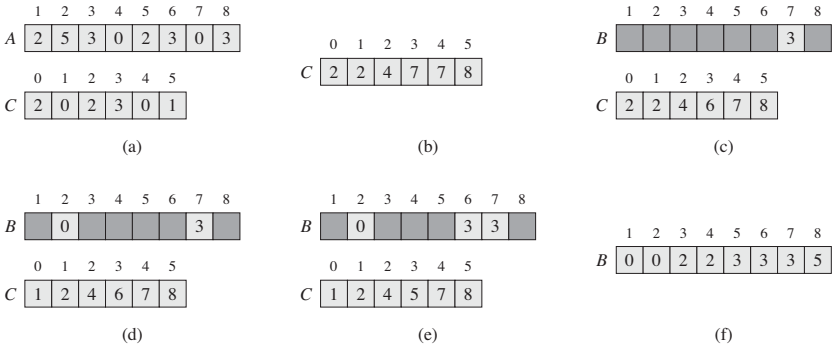


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Correctitud de ordenamiento por conteo

Al finalizar los tres primeros lazos, $C[A[i]]$ contiene el número de elementos menores o iguales a $A[i]$ en el arreglo $A[1 \ldots n]$, para $n = A.\text{length}$

Las $k = C[A[i]]$ ocurrencias de $A[i]$ deben aparecer en el arreglo de salida en las posiciones:

$$C[A[i]],\ C[A[i]] - 1,\ \ldots,\ C[A[i]] - k + 1$$

Por otro lado, las $k = C[A[i]]$ ocurrencias de $a = A[i]$ aparecen en k posiciones $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ en $A[1 \ldots n]$

El último lazo coloca $A[i_k]$ en la posición $C[a]$, $A[i_{k-1}]$ en la posición $C[a] - 1$, y así sucesivemante hasta colocar $A[i_1]$ en $C[a] - k + 1$

□

Análisis de ordenamiento por conteo

`Counting-Sort` ejecuta 4 lazos donde cada iteración toma **tiempo constante:**

- el primer lazo ejecuta $k + 1$ iteraciones
- el segundo lazo ejecuta n iteraciones donde $n = A.length$
- el tercer lazo ejecuta k iteraciones
- el último lazo ejecuta n iteraciones

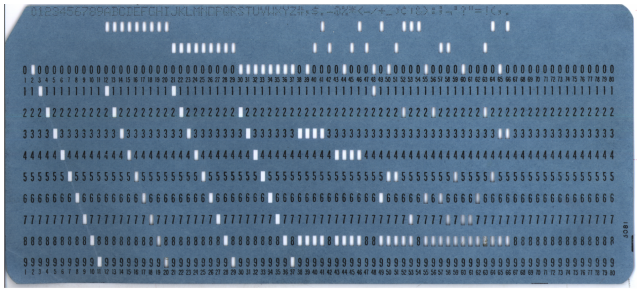
El tiempo total es $\Theta(n + k)$

Pare el caso típico $k = O(n)$, `Counting-Sort` toma tiempo $\Theta(n)$

Radix sort

Radix sort es el algoritmo de ordenamiento utilizado en las antiguas computadoras para ordenar tarjetas

Cada tarjeta contiene 80 columnas y 12 filas. En el caso más sencillo, cada columna contiene uno de 12 símbolos distintos, denotado por una perforación en la fila correspondiente



https://en.wikipedia.org/wiki/IBM_card_sorter

Radix sort

Un **ordenador de tarjetas** (sorting machine) es una máquina que clasifica las tarjetas según las perforaciones. El ordenador tiene:

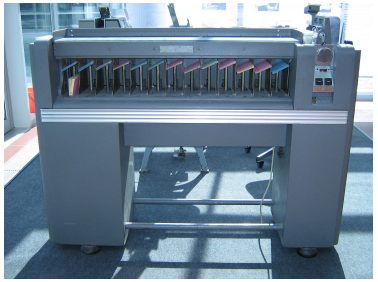
- una bandeja de entrada donde se colocan las tarjetas a clasificar
- 13 bandejas de salida donde se distirbuyen las tarjetas en la bandeja de entrada

El ordenador considera sólo una **columna seleccionada** al procesar las tarjetas

En el caso más sencillo, cada tarjeta es colocada en la bandeja correspondiente a la perforacion que contiene en la columna seleccionada con una bandeja adicional para errores y tarjetas sin perforación

Ordenador y perforador de tarjetas IBM

IBM 082



https://en.wikipedia.org/wiki/IBM_card_sorter

IBM 029

http://q7.neurotica.com/Oldtech/Keypunch/IBM029.html

Radix sort

Para ordenar las tarjetas por el valor númerico perforado, el operador:

- clasifica las tarjetas por el dígito menos significativo
- concatena el contenido de cada bandeja en la bandeja de entrada, por orden de dígito
- clasifica las tarjetas de nuevo por el segundo dígito menos significativo
- repite hasta clasificar y concatenar por el dígito más significativo

Radix sort en acción

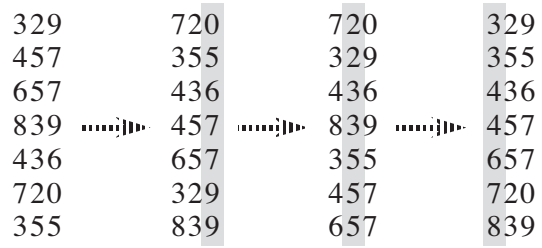


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Radix sort

Input: arreglo $A[1 \ldots n]$ con enteros no negativos de d-dígitos y d

Output: arreglo A reordenado de menor a mayor

```
1  Radix-Sort(array A, int d)
2      for i = 1 to d
3          Ordenar de forma estable el arreglo A por el i-ésimo dígito
```

Nota: el dígito menos significativo es el primer dígito y el más significativo es el d-ésimo dígito

Correctitud de radix sort (1 de 2)

Considere un arreglo A con n enteros no negativos de d dígitos c/u donde $A[i][j]$ denota el j-ésimo dígito del i-ésimo entero

Probaremos que al comenzar la k-ésima iteración, los enteros están ordenados por los primeros $k-1$ dígitos

Así, al finalizar, los enteros estarán ordenados por los d-dígitos

Correctitud de radix sort (2 de 2)

Caso base: al comenzar la primera iteración $k - 1 = 0$ y los enteros no están ordenados

Tesis: suponga la hipótesis y considere la k-ésima iteración durante la cual los enteros son ordenados de **forma estable** por el k-ésimo dígito

Sean z y z' dos enteros en el arreglo tal que $z[1 \ldots k] < z'[1 \ldots k]$

- Si $z[k] < z'[k]$ entonces z aparece antes que z' al finalizar la iteración porque los números son ordenados por el k-ésimo dígito
- Si $z[k] = z'[k]$ y $z[1 \ldots k-1] < z'[1 \ldots k-1]$, entonces al inicio de la iteración z estaba antes que z' (por HI). Como el algoritmo es estable, al finalizar la iteración z está antes que z' en el arreglo $\square$

Análisis de radix sort

`Radix-Sort` realiza d ordenamientos de n elementos cada uno

Cada ordenamiento puede hacerse con `Counting-Sort` en tiempo $O(n + k) = O(n)$ (con $k = 10$) ya que es sobre los dígitos de cada número

El tiempo total de `Radix-Sort` es $\Theta(dn)$

Dígitos de multiple precisión

Para ordenar n enteros de b bits c/u (donde cada entero pertenece a $[0, 2^b)$), podemos definir **nuevos dígitos** agrupando bits en bloques de r bits (donde cada dígito $\in [0, 2^r - 1]$) y luego aplicar `Radix-Sort`

En este caso tenemos $d = \lceil b/r \rceil$ dígitos por número, y si utilizamos `Counting-Sort`, podemos ordenar todos los números en tiempo:

$$\Theta(d(n+k)) \;=\; \Theta(\lceil b/r \rceil (n + 2^r - 1)) \;=\; \Theta((b/r)(n + 2^r))$$

¿Cuál es el mejor r para n y b dados?

• Si $b \leq \lfloor \log_2 n \rfloor$, $n + 2^r = \Theta(n)$ para cualquier $r \leq b$, y `Radix-Sort` corre en tiempo $\Theta(n)$ para $r = b$ **(optimalidad asintótica)**

• Si $b > \lfloor \log_2 n \rfloor$, el mejor r es $r = \lfloor \log_2 n \rfloor$ y `Radix-Sort` corre en tiempo $\Theta(bn/\log n)$

Consideraciones prácticas

¿És `Radix-Sort` mejor que `Quicksort` en la práctica?

La respuesta depende de muchos factores

Aún cuando `Radix-Sort` puede correr en tiempo $\Theta(n)$ mientras que `Quicksort` corre en tiempo esperado $\Theta(n \log n)$, las constantes escondidas para `Radix-Sort` son peores que para `Quicksort`

Además el algoritmo `Counting-Sort` utilizado por `Radix-Sort` requiere de memoria adicional

Por lo tanto, en algunos casos `Quicksort` puede correr más rápido que `Radix-Sort`

Bucket sort

A diferencia de ordenamiento por conteo y radix sort, bucket sort asume que la entrada son **números reales** que han sido generados de forma **uniforme** sobre el intervalo $[0, 1)$

Bucket sort divide el intervalo $[0, 1)$ en n subintervalos o **buckets** de igual tamaño y distribuye la entrada de n números sobre los buckets

Luego cada bucket es ordenado de forma **independiente** y el contenido de estos es concatenado en la salida

Bucket sort en acción

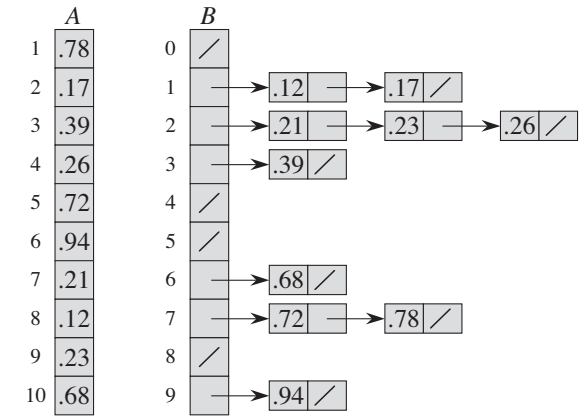


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Bucket sort

Input: arreglo A con enteros en $\{0, \ldots, k\}$, arreglo de salida B y k

Output: arreglo B con elementos de A ordenados de menor a mayor

```
1  Bucket-Sort(array A)
2      n = A.length
3      let B[0..n-1] be a new array
4      for i = 0 to n - 1
5          B[i] = empty list
6
7      for i = 1 to n
8          Insertar A[i] en la lista B[⌊n*A[i]⌋]
9
10     for i = 0 to n - 1
11         Ordenar lista B[i] con Insertion-Sort
12
13     Concatenar listas B[0], B[1], ..., B[n-1] en orden
```


Correctitud de bucket sort

Considere dos elementos $A[i]$ y $A[j]$ en el arreglo de entrada

Sin perdida de generalidad asuma que $A[i] \leq A[j]$

Consideramos los casos $\lfloor nA[i] \rfloor < \lfloor nA[j] \rfloor$ y $\lfloor nA[i] \rfloor = \lfloor nA[j] \rfloor$:

• **Caso** $\lfloor nA[i] \rfloor < \lfloor nA[j] \rfloor$**:** el lazo 7–8 pone al elemento $A[i]$ en un bucket anterior al bucket para $A[j]$. Entonces, $A[i]$ aparece antes que $A[j]$ en la salida

• **Caso** $\lfloor nA[i] \rfloor = \lfloor nA[j] \rfloor$**:** el lazo 7-8 coloca a ambos $A[i]$ y $A[j]$ en el mismo bucket $B[k]$. El bucket $B[k]$ es ordenado en el lazo 10–11, y entonces $A[i]$ aparece antes que $A[j]$ en la salida □

Análisis de bucket sort (1 de 4)

Observe que excluyendo el tiempo tomado por `Insertion-Sort`, `Bucket-Sort` toma tiempo lineal. Así debemos cuantificar cuanto tiempo toman las n llamadas a `Insertion-Sort`

Sea n_i la v.a. para el número de elementos colocados en el bucket i. Como `Insertion-Sort` toma tiempo cuadrático,

$$T(n) \;=\; \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Ahora tomaremos valor esperado a ambos lados

Análisis de bucket sort (2 de 4)

$$\begin{aligned}
\mathbb{E}[T(n)] &= \mathbb{E}\left[\Theta(n) + \textstyle\sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \mathbb{E}[\Theta(n)] + \textstyle\sum_{i=0}^{n-1} \mathbb{E}\left[O(n_i^2)\right] \\
&= \Theta(n) + \textstyle\sum_{i=0}^{n-1} O(\mathbb{E}[n_i^2]) \\
&= \Theta(n) + \textstyle\sum_{i=0}^{n-1} O(2 - \frac{1}{n}) \\
&= \Theta(n) + n\, O(2 - \tfrac{1}{n}) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}$$

donde $\mathbb{E}[n_i^2] = O(2 - \frac{1}{n})$ (ver próximos dos slides)

Análisis de bucket sort (3 de 4)

Es lógico que el valor de $\mathbb{E}[n_i^2]$ **no dependa** de i ya que todos los buckets son del mismo tamaño y cada elemento pertenece a cada bucket con igual probabilidad (por la suposición sobre la entrada)

Defina la v.a. indicadora $X_{ij} = \mathbb{I}\{A[j] \text{ cae en el bucket } i\}$ para $0 \leq i < n$ y $1 \leq j \leq n$. Entonces $n_i = \sum_{j=1}^{n} X_{ij}$

Calculamos $\mathbb{E}[n_i^2]$:

$$\begin{aligned}
\mathbb{E}[n_i^2] &= \mathbb{E}\left[\left(\textstyle\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathbb{E}\left[\textstyle\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathbb{E}\left[\textstyle\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{1\leq j<k\leq n} X_{ij}X_{ik}\right] \\
&= \textstyle\sum_{j=1}^{n} \mathbb{E}[X_{ij}^2] + 2\sum_{1\leq j<k\leq n} \mathbb{E}[X_{ij}X_{ik}]
\end{aligned}$$

Análisis de bucket sort (4 de 4)

$$\mathbb{E}[X_{ij}^2] \;=\; 1^2 \cdot \frac{1}{n} \;+\; 0^2 \cdot \left(1 - \frac{1}{n}\right) \;=\; \frac{1}{n}$$

Cuando $j \neq k$, X_{ij} es independiente de X_{ik},

$$\mathbb{E}[X_{ij}X_{ik}] \;=\; \mathbb{E}[X_{ij}] \cdot \mathbb{E}[X_{ik}] \;=\; \frac{1}{n} \cdot \frac{1}{n} \;=\; \frac{1}{n^2}$$

Entonces,

$$\begin{aligned}
\mathbb{E}[n_i^2] &= \textstyle\sum_{j=1}^{n} \mathbb{E}[X_{ij}^2] + 2\sum_{1\leq j<k\leq n} \mathbb{E}[X_{ij}X_{ik}] \\
&= n \cdot \frac{1}{n} + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \;=\; 2 - \frac{1}{n}
\end{aligned}$$

□

Resumen

- Al utilizar información adicional de los elementos es posible diseñar algoritmos de ordenamiento que corren en tiempo $o(n \log n)$
- `Counting-Sort` asume que los elementos a ordenar son n enteros en el rango $\{0, \ldots, d\}$ y corre en tiempo $\Theta(n + d)$
- `Radix-Sort` asume que los elementos a ordenar son n enteros de a lo sumo d dígitos y corre en tiempo $\Theta(dn)$
- `Bucket-Sort` asume que los elementos a ordenar son n números reales en $[0, 1)$ generados de forma uniforme e independiente, y corre en tiempo esperado $\Theta(n)$

Ejercicios (1 de 6)

1. (8.2-1) Corra `Counting-Sort` sobre la entrada $\langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$
2. (8.2-2) Muestre que `Counting-Sort` es un algoritmo estable
3. (8.2-3) Suponga que cambiamos el último lazo en `Counting-Sort` de forma que comienze en $j = 1$ y termine en $j = A.\mathit{length}$

   Muestre que el algoritmo sigue siendo correcto pero no estable
4. (8.2-4) Diseñe un algoritmo que dados n enteros en $\{0, 1 \ldots, k\}$, **preprocese la entrada** y luego sea capaz de responder en **tiempo constante** cualquier pregunta ("query") del tipo:

   ¿Cuántos elementos en la entrada existen en el rango $[a, b)$?

   El tiempo de procesamiento debe ser $O(n + k)$

Ejercicios (2 de 6)

5. (8.3-4) Utilizando `Radix-Sort` muestre como ordenar n enteros en $\{0, \ldots, n^3 - 1\}$ en tiempo $\Theta(n)$
6. (8.4-1) Corra `Bucket-Sort` sobre $\langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$
7. (8.4-2) ¿Por qué `Bucket-Sort` toma tiempo $\Theta(n^2)$ en el peor caso?

   Modifique `Bucket-Sort` para que corra en tiempo $\Theta(n \log n)$ en el peor caso.
8. `Bucket-Sort` divide el intervalo $[0, 1)$ en n subintervalos tal que la probabilidad de que un número caiga en un subintervalo es $1/n$. ¿Qué sucede con el tiempo de corrida si modificamos `Bucket-Sort` para que divida $[0, 1)$ en $n/2$ subintervalos? ¿Qué sucede con el tiempo de corrida si modificamos `Bucket-Sort` para que divida $[0, 1)$ en $\log n$ subintervalos?

Ejercicios (3 de 6)

9. (8.4-4)* Considere n puntos $\{p_i\}_{i=1}^n$ en el circulo unitario; i.e. para $1 \leq i \leq n$, $p_i = (x_i, y_i)$ con $\sqrt{x_i^2 + y_i^2} \leq 1$. Suponga que los puntos son distribuidos uniforme e independientemente (i.e. la probabilidad de que un punto p_i caiga en una región R dentro del circulo unitario es igual a Area$(R)/\pi$

   Diseñe un algoritmo que corra en tiempo lineal **esperado** para ordenar los puntos p_i por sus distancias $d_i = \sqrt{x_i^2 + y_i^2}$ al origen.

   (**Ayuda:** diseñe n buckets tal que la probabilidad de que un punto cualquiera caiga en un bucket cualquiera sea $1/n$.)
10. (8.4-5)* La función de probabilidad $p(x)$ de una variable aleatoria X se define por $p(x) = P(X \leq x)$. Suponga que tenemos n muestras $x_1, \ldots, x_n$ obtenidas de forma independiente de una v.a. X con función de probabilidad continua la cual es computable en tiempo constante. Diseñe un algoritmo para ordenar los n números en tiempo lineal esperado

Ejercicios (4 de 6)

11. (8-2) Considere un arreglo con n registros con claves en $\{0, 1\}$, y las siguientes propiedades para algoritmos que ordenen tales registros:

    P1. El algoritmo corre en tiempo $O(n)$
    P2. el algoritmo es estable
    P3. el algoritmo utiliza a lo sumo espacio adicional constante

    a. De un algoritmo que satisfaga P1 y P2
    b. De un algoritmo que satisfaga P1 y P3
    c. De un algoritmo que satisfaga P2 y P3
    d. ¿Se puede utilizar alguno de los algoritmos en (a)–(c) como el algoritmo de ordenamiento en `Radix-Sort` para ordenar n registros con claves de b bits en tiempo $O(bn)$? Explique
    e. Suponga que los registros tienen claves en $\{1, \ldots, k\}$. Modifique `Counting-Sort` para ordenar los n registros "in place" en tiempo $O(n + k)$. (Puede utilizar $O(k)$ espacio adicional.) ¿Es el algoritmo resultante un algoritmo estable?

Ejercicios (5 de 6)

12. (8-3)* Ordenamiento de elementos de tamaño variable

    a. Considere un arreglo de enteros de multiple precisión (longitud variable) donde dos enteros distintos no tienen que ser del mismo tamaño. El número total de dígitos sobre todos los enteros es n. Diseñe un algoritmo que ordene los enteros en tiempo $O(n)$

    b*. Considere un arreglo de "strings" donde strings distintos no tienen que ser del mismo tamaño. El número total de caracteres sobre todos los strings es n. Diseñe un algoritmo que ordene los strings de forma alfabética en tiempo $O(n)$. El algoritmo no tiene que ordenar "in place". Puede recibir un arreglo B de salida

    (El orden alfabético es el usan los diccionarios; e.g. a $<$ ab $<$ b)

Ejercicios (6 de 6)

13. (8-3)* Jarras de agua

    Considere n jarras rojas y n jarras azules de agua de formas distintas. Las n jarras rojas de agua son de capacidades diferentes, así como las azules. Para cada jarra roja existe una azul de igual capacidad, y vice versa.

    La tarea es aparear las jarras rojas con las azules de igual capacidad; i.e. encontrar n pares de jarras rojas con azules. La única operación que puede hacerse es comparar una jarra roja con una azul. La comparación se hace virtiendo agua de una jarra a la otra. La comparación indica si ambas jarras tienen la misma capacidad, o si una tiene más que la otra.

    a. Diseñe un algoritmo que realize $\Theta(n^2)$ comparaciones

    b. Pruebe una cota inferior de $\Omega(n \log n)$ comparaciones. (**Ayuda:** si puede resolver el problema con $o(n \log n)$ comparaciones, puede ordenar n elementos con $o(n \log n)$ comparaciones)

    c*. De un **algoritmo randomizado** que haga $O(n \log n)$ comparaciones esperadas. ¿Cuántas comparaciones realiza su algoritmo en el peor caso?