# Decrementa y vencerás

Dr. Eduardo A. Rodríguez Tello

Cinvestav-Tamaulipas

19 de febrero de 2018

1. Decrementa y vencerás
   - Tipos de algoritmos decrementa y vencerás
   - Ordenamiento por inserción
   - Algoritmos para generar objetos combinatorios

# Decrementa y vencerás

## Algoritmo decrementa y vencerás

1. Reduce la instancia del problema a la instancia más pequeña del mismo problema
2. Resuelve la instancias más pequeña
3. Extiende la solución de la instancias más pequeña para obtener una solución del problema original

- Puede ser implementado de arriba-abajo (*top-down*), o de abajo-arriba (*bottom-up*)
- También se conoce como enfoque inductivo o incremental

Cinvestav

# Decrementa y vencerás

Existen tres tipos de algoritmos decrementa y vencerás:

1. **Decremento con una constante** (usualmente 1):
    - Ordenamiento por inserción
    - Ordenamiento topológico
    - Algoritmos para generar permutaciones, subconjuntos
2. **Decremento con factor constante** (usualmente la mitad)
    - Búsqueda binaria
    - Exponenciación por cuadrados
    - Multiplicación por el método ruso
3. **Decremento de tamaño variable**
    - Algoritmo de Euclides
    - Selección por partición
    - Juego de tipo Nim

# Decrementa y vencerás, ¿Diferencia?

Considere el problema de exponenciación: calcular $f(n) = a^n$

- Fuerza bruta:

$$a^n = \underbrace{a * \cdots * a}_{n \text{ veces}}$$

- Decrementa en uno:

$$a^n = \begin{cases} a^{n-1} \cdot a & \text{si } n > 0, \\ 1 & \text{si } n = 0, \end{cases}$$

- Decrementa en un factor constante:

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ es par y positivo}, \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{si } n \text{ es impar}, \\ 1 & \text{si } n = 0, \end{cases}$$

1. Decrementa y vencerás
   - Tipos de algoritmos decrementa y vencerás
   - Ordenamiento por inserción
   - Algoritmos para generar objetos combinatorios

# Ordenamiento por inserción

- Para ordenar un arreglo $A[0 \ldots n-1]$, ordena recursivamente $A[0 \ldots n-2]$ e inserta después $A[n-1]$ en su posición correcta dentro del sub-arreglo $A[0 \ldots n-2]$
- Usualmente se implementa *bottom-up* (no recursivo)

$$A[0] \leq \ldots \leq A[j] < A[j+1] \leq \ldots \leq A[i-1] \quad | \quad A[i] \ldots A[n-1]$$

| | | | | | | |
|---|---|---|---|---|---|---|
| 89 \| | **45** | 68 | 90 | 29 | 34 | 17 |
| 45 | 89 \| | **68** | 90 | 29 | 34 | 17 |
| 45 | 68 | 89 \| | **90** | 29 | 34 | 17 |
| 45 | 68 | 89 | 90 \| | **29** | 34 | 17 |
| 29 | 45 | 68 | 89 | 90 \| | **34** | 17 |
| 29 | 34 | 45 | 68 | 89 | 90 \| | **17** |
| 17 | 29 | 34 | 45 | 68 | 89 | 90 |

# Ordenamiento por inserción

**ALGORITHM** *InsertionSort*($A[0..n-1]$)

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of $n$ orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
**for** $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    **while** $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

# Ordenamiento por inserción, Eficiencia temporal

**ALGORITHM** *InsertionSort*($A[0..n-1]$)
  //Sorts a given array by insertion sort
  //Input: An array $A[0..n-1]$ of $n$ orderable elements
  //Output: Array $A[0..n-1]$ sorted in nondecreasing order
  **for** $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    **while** $j \geq 0$ **and** $A[j] > v$ **do**
      $A[j+1] \leftarrow A[j]$
      $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

## Peor caso

# Ordenamiento por inserción, Eficiencia temporal

**ALGORITHM** *InsertionSort*$(A[0..n-1])$
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of $n$ orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
**for** $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow A[i]$
  $j \leftarrow i-1$
  **while** $j \geq 0$ **and** $A[j] > v$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
  $A[j+1] \leftarrow v$

## Peor caso

Arreglo ordenado decrecientemente. La operación básica es la comparación $A[j] > v$

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

# Ordenamiento por inserción, Eficiencia temporal

**ALGORITHM** *InsertionSort*$(A[0..n-1])$
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of $n$ orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
**for** $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow A[i]$
  $j \leftarrow i-1$
  **while** $j \geq 0$ **and** $A[j] > v$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
  $A[j+1] \leftarrow v$

## Mejor caso

# Ordenamiento por inserción, Eficiencia temporal

**ALGORITHM** *InsertionSort*($A[0..n-1]$)
    //Sorts a given array by insertion sort
    //Input: An array $A[0..n-1]$ of $n$ orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    **for** $i \leftarrow 1$ **to** $n-1$ **do**
        $v \leftarrow A[i]$
        $j \leftarrow i-1$
        **while** $j \geq 0$ **and** $A[j] > v$ **do**
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j-1$
        $A[j+1] \leftarrow v$

## Mejor caso

Arreglo ordenado crecientemente.

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

# Ordenamiento por inserción, Eficiencia temporal

**ALGORITHM** *InsertionSort*$(A[0..n-1])$

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of $n$ orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
**for** $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow A[i]$
  $j \leftarrow i-1$
  **while** $j \geq 0$ **and** $A[j] > v$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
  $A[j+1] \leftarrow v$

## Caso promedio

# Ordenamiento por inserción, Eficiencia temporal

```
ALGORITHM  InsertionSort(A[0..n − 1])
    //Sorts a given array by insertion sort
    //Input: An array A[0..n − 1] of n orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 1 to n − 1 do
        v ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > v do
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← v
```

## Caso promedio

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

También es rápido en arreglos casi ordenados

# Ordenamiento por inserción, Ventajas

- Su eficiencia promedio junto con la excelente eficiencia en arreglos casi ordenados permite al *ordenamiento por inserción* destacar frente a otros algoritmos elementales de ordenamiento (por selección y de burbuja) aplicados a conjuntos de datos relativamente pequeños
- Su extensión conocida como ordenamiento Shell [Shell 1959] da incluso mejores resultados al aplicarlo a archivos relativamente grandes
- Eficiencia espacial: in-situ (*in-place*), i.e., está acotada por una función constante $O(1)$
- Es *estable*, i.e., no cambia el orden relativo de los elementos con igual llave

# Generación de permutaciones

Algoritmo de Heap [Heap, 1963], mínimo cambio - decremento en uno

**Procedure** $\text{generatePermutations}(n, A)$

**Require:** Longitud de las permutaciones $n$; arreglo $A$ de tamaño $m$ donde $m \geq n$

```
if n == 1 then
    print(A)
else
    for i ← 1; i ≤ n; i ← i + 1 do
        generatePermutations(n − 1, A)
        if n es impar then
            j ← 1
        else
            j ← i
        end if
        swap(A[j], A[n])
    end for
end if
```

Cinvestav

# Generación de permutaciones

Podemos observar un buen ejemplo del funcionamiento de este algoritmo en la siguiente URL:

`http://en.wikipedia.org/wiki/Heap's_algorithm`

# Algoritmo de Heap, Análisis

- $\mathrm{swap}$ es la operación básica. El número de operaciones realizadas para un arreglo de $n$ elementos es $S(n)$:

$$S(1) = 0$$

$$S(n) = \sum_{i=1}^{n} [S(n-1) + 1]$$

$$S(n) = nS(n-1) + n \qquad \text{para } n > 1$$

# Algoritmo de Heap, Análisis

- swap es la operación básica. El número de operaciones realizadas para un arreglo de $n$ elementos es $S(n)$:

$$S(1) = 0$$

$$S(n) = \sum_{i=1}^{n} [S(n-1) + 1]$$

$$S(n) = nS(n-1) + n \qquad \text{para } n > 1$$

Resolviendo la relación de recurrencia mediante sustitución hacia atrás:

$$S(n) = n^{n-1}S(1) + \sum_{k=1}^{n-1} n^k$$

Como $S(1) = 0$ sólo queda el término de la derecha:

$$S(n) = \sum_{k=1}^{n-1} n^k \approx n!(e-1) - 1 \in \Theta(n!)$$

# Ejercicio

Algoritmo de Heap [Heap, 1963], mínimo cambio - decremento en uno

**Procedure** generatePermutations($n$, $A$)

**Require:** Longitud de las permutaciones $n$; arreglo $A$ de tamaño $m$ donde $m \geq n$

```
if n == 1 then
    print(A)
else
    for i ← 1; i ≤ n; i ← i + 1 do
        generatePermutations(n − 1, A)
        if n es impar then
            swap(A[1], A[n])
        else
            swap(A[i], A[n])
        end if
    end for
end if
```

- Dado $A = \{1, 2, \ldots, n\}$, y $n = 4$ trace la salida del algoritmo
- ¿Cómo podría probarse la correctitud (*correctness*) del algoritmo?

# Generación de permutaciones

Respuesta:

| | | | | | |
|---|---|---|---|---|---|
| 1234 | 2134 | 3124 | 1324 | 2314 | 3214 |
| 4231 | 2431 | 3421 | 4321 | 2341 | 3241 |
| 4132 | 1432 | 3412 | 4312 | 1342 | 3142 |
| 4123 | 1423 | 2413 | 4213 | 1243 | 2143 |

# Otros algoritmos para generar permutaciones

- Steinhaus-Johnson-Trotter (Johnson-Trotter)
- Lexicographic-order

# Algoritmo Steinhaus-Johnson-Trotter

- Podemos generar las permutaciones en un mismo orden sin generar explícitamente permutaciones de tamaño más pequeño
- Para ello asociamos una dirección a cada elemento $k$ (flecha)
- Un elemento $k$ es movible si su flecha apunta a un número adyacente más pequeño
- Ejemplo con $n = 3$: $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$
- 3 y 4 son movibles mientras que 1 y 2 no lo son

# Algoritmo Steinhaus-Johnson-Trotter

**ALGORITHM** *JohnsonTrotter*$(n)$

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer $n$
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
**while** the last permutation has a mobile element **do**
    find its largest mobile element $k$
    swap $k$ with the adjacent element $k$'s arrow points to
    reverse the direction of all the elements that are larger than $k$
    add the new permutation to the list

- Ejemplo con $n = 3$

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}} \quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2} \quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{2}} \quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}.$$

# Algoritmo Steinhaus-Johnson-Trotter

- Este algoritmo es uno de los más eficientes para generar permutaciones
- Puede ser implementado para correr en tiempo proporcional al número de permutaciones, i.e., $\Theta(n!)$

# Algoritmo Lexicographic-order

- Observemos que el orden en que genera las permutaciones el algoritmo Steinhaus-Johnson-Trotter no es el más natural

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}} \quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2} \quad \overleftarrow{\mathbf{3}}\,\overleftarrow{1}\,\overleftarrow{2} \quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}.$$

- Por ejemplo, sería deseable que la permutación $n(n-1)\ldots 1$ fuera la última de la lista
- Esto sería el caso si se listaran en orden ascendente, también conocido como lexicográfico
- Ejemplo con $n = 3$:

$$123 \quad 132 \quad 213 \quad 231 \quad 312 \quad 321$$

# Algoritmo Lexicographic-order

- Cómo podemos generar la permutación que sigue de $a_1, a_2, \ldots, a_{n-1}, a_n$ en el orden lexicográfico
- Si $a_{n-1} < a_n$, lo cual es el caso para la mitad de todas las permutaciones, podemos simplemente intercambiar estos dos últimos elementos
- Por ejemplo, $123$ es seguido de $132$

# Algoritmo Lexicographic-order

- Si $a_{n-1} > a_n$, por ejemplo $(5, 1, 7, 6, 3, 9, 8, 4, 2)$
  - Encontramos el sufijo decreciente más largo de la permutación $a_{i+1} > a_{i+2} > \ldots > a_n$ (pero $a_i < a_{i+1}$). Iniciando al final tenemos que $4 > 2$, $8 > 4$, $9 > 8$, y finalmente $3 < 9$.
  - Por lo tanto el pivote es $i = 5$ ($a_i = 3$ y $a_{i+1} = 9$): $(5, 1, 7, 6, 3, 9, 8, 4, 2)$
  - Iniciando en $i + 1$ observamos que $3 < 9$, $3 < 8$, $3 < 4$, pero $3 > 2$. El ultimo valor que es todavía mayor que $3$ es $4$, y su índice es $j = 8$.
  - Intercambiamos $a_i$ y $a_j$ (incrementando $a_i$), para obtener: $(5, 1, 7, 6, 4, 9, 8, 3, 2)$
  - Invertimos el nuevo sufijo (desde $a_{i+1}$ hasta $a_n$) para obtener el resultado: $(5, 1, 7, 6, 4, 2, 3, 8, 9)$

# Algoritmo Lexicographic-order

**ALGORITHM** *LexicographicPermute*($n$)

//Generates permutations in lexicographic order
//Input: A positive integer $n$
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
**while** last permutation has two consecutive elements in increasing order **do**
    let $i$ be its largest index such that $a_i < a_{i+1}$    //$a_{i+1} > a_{i+2} > \cdots > a_n$
    find the largest index $j$ such that $a_i < a_j$    //$j \geq i + 1$ since $a_i < a_{i+1}$
    swap $a_i$ with $a_j$    //$a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
    reverse the order of the elements from $a_{i+1}$ to $a_n$ inclusive
    add the new permutation to the list

# Generación de subconjuntos

- El conjunto potencia de un conjunto $A = \{a_1, \ldots, a_n\}$ es el conjunto de todos los subconjuntos de $A$, incluyendo el conjunto vacío y al mismo $A$.
- Ejemplos:

| $n$ | | | | subsets | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | $\varnothing$ | | | | | | | |
| 1 | $\varnothing$ | $\{a_1\}$ | | | | | | |
| 2 | $\varnothing$ | $\{a_1\}$ | $\{a_2\}$ | $\{a_1, a_2\}$ | | | | |
| 3 | $\varnothing$ | $\{a_1\}$ | $\{a_2\}$ | $\{a_1, a_2\}$ | $\{a_3\}$ | $\{a_1, a_3\}$ | $\{a_2, a_3\}$ | $\{a_1, a_2, a_3\}$ |

- Para un conjunto con $n$ elementos es posible formar $2^n$ subconjuntos
- ¿Cómo generarlos?

# Generación de subconjuntos

Código binario reflejado (Código Gray):

- Algoritmo de mínimo cambio para generar las $2^n$ cadenas binarias correspondientes a todos los subconjuntos de un conjunto de $n$ elementos ($n > 0$)
- Ventaja: cada cadena difiere de su predecesora sólo en un bit
- Suponga el caso $n = 3$

| bit strings | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| subsets | $\varnothing$ | $\{a_3\}$ | $\{a_2\}$ | $\{a_2, a_3\}$ | $\{a_1\}$ | $\{a_1, a_3\}$ | $\{a_1, a_2\}$ | $\{a_1, a_2, a_3\}$ |

# Generación de subconjuntos

**ALGORITHM** *BRGC(n)*

//Generates recursively the binary reflected Gray code of order $n$
//Input: A positive integer $n$
//Output: A list of all bit strings of length $n$ composing the Gray code
**if** $n = 1$ make list $L$ containing bit strings 0 and 1 in this order
**else** generate list $L1$ of bit strings of size $n-1$ by calling $BRGC(n-1)$
    copy list $L1$ to list $L2$ in reversed order
    add 0 in front of each bit string in list $L1$
    add 1 in front of each bit string in list $L2$
    append $L2$ to $L1$ to get list $L$
**return** $L$

Ejemplo: generar el código binario reflejado para $n = 4$.
Como el código Gray para $n = 3$ es: $000\ 001\ 011\ 010\ 110\ 111\ 101\ 100$, entonces el algoritmo haría lo siguiente:

$L1$ 000 001 011 010 110 111 101 100

$L2$ 100 101 111 110 010 011 001 000

$L$ 0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111 1110 1010 1011 1001 1000