

CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Nociones básicas

© 2018 Blai Bonet

Objetivos

- Concepto de algoritmo y modelo computacional
- Complejidad en tiempo y espacio de algoritmos
- Repasar conceptos de crecimiento de funciones y notación asintótica
- Búsqueda lineal y binaria sobre un arreglo
- Ordenamiento por inserción y su análisis

© 2018 Blai Bonet

Algoritmos

Un algoritmo es un **procedimiento** para resolver una tarea específica que está descrito en un **lenguaje de programación**. El algoritmo se ejecuta sobre un **modelo computacional**

El algoritmo resuelve la tarea para una **instancia** dada como **entrada**. La **salida** del algoritmo es la solución de la tarea sobre la instancia

- La **longitud** de la entrada es medida en **bits**
- El **tiempo de ejecución** es medido en unidades de tiempo fijas como segundos
- La **cantidad de memoria** utilizada por el algoritmo (adicional a la entrada) es medida en **bits**

© 2018 Blai Bonet

Caja negra



Modelo computacional: RAM

Modelo: random-access machine (RAM) con único procesador secuencial

Tipos básicos: enteros y punto flotante de **precisión acotada**

(Asumimos que todas las operaciones aritméticas y punto flotante toman **tiempo constante** lo que implica que el **tamaño de palabra** es suficiente para guardar las cantidades manejadas. No podemos asumir **precisión arbitraria** porque entonces podríamos guardar cantidades arbitrarias de información en una celda de memoria o registro.)

Memoria: computador tiene infinitas celdas de memoria. Las celdas pueden direccionarse directamente (random-access)

Complejidad en tiempo y espacio

Considere un algoritmo A

El **tiempo de ejecución** de A es una **función** T_A tal que $T_A(\omega)$ es el número de unidades de tiempo que A toma sobre la entrada es ω

El **consumo de memoria** de A es una **función** M_A tal que $M_A(\omega)$ es el número de bits de memoria que A utiliza sobre la entrada es ω

En el curso nos enfocamos en el **tiempo de ejecución** ya que:

- el consumo de memoria está acotado por el tiempo, $M_A \leq T_A$: en X unidades de tiempo solo pueden accesarse a lo sumo X celdas de memoria
- los algoritmos que veremos tienen poco consumo de memoria

Consumo de tiempo en el peor caso

Considere un algoritmo A con función de tiempo T_A

La función de tiempo en el **peor caso** para A mide para cada entero n , el mayor tiempo que toma A en una entrada de tamaño n

Formalmente, la función de tiempo en el peor caso para A es una función $T_A : \mathbb{N} \rightarrow \mathbb{N}$ dada por

$$T_A(n) = \max \{ T_A(\omega) : |\omega| = n \}$$

Nos interesa conocer que tan rápido crece $T_A(n)$ cuando $n \rightarrow \infty$

Consumo de tiempo en el caso promedio

Aunque importante, el peor caso es una **medida pesimista** que puede reflejar incorrectamente el desempeño del algoritmo en la práctica

Una medida mas realista es el desempeño en el **caso promedio**

Para hablar de caso promedio necesitamos conocer como se **distribuyen** las posibles entradas al algoritmo

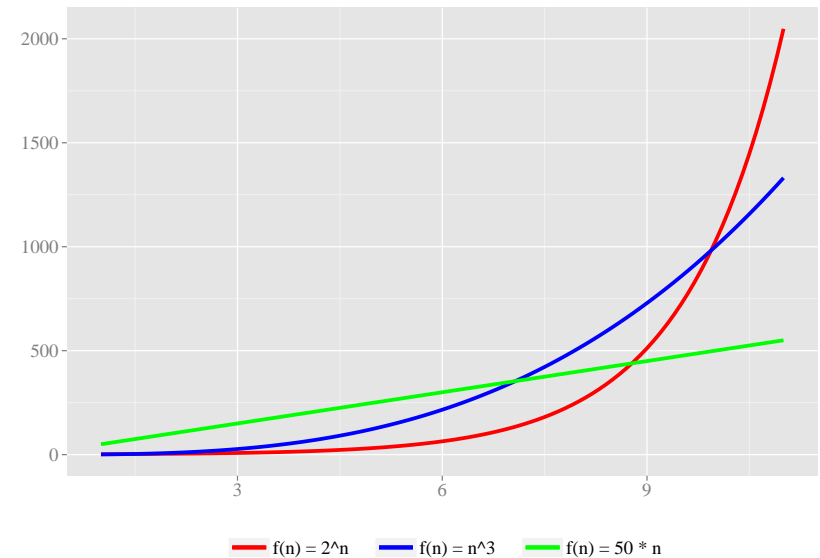
Para un n fijo, asumimos una **distribución uniforme** sobre las entradas de tamaño n : **cada entrada es igualmente probable**

El tiempo promedio sobre entradas de tamaño n para A es:

$$\frac{1}{m} \sum_{\omega: |\omega|=n} T_A(\omega)$$

donde m es el número de entradas de tamaño n

Crecimiento de funciones



Notación asintótica

- Dominancia: $o(\cdot)$ (o -pequeña) y $\omega(\cdot)$ (ω -pequeña)
- Cotas superiores: $O(\cdot)$ (O -grande)
- Cotas inferiores: $\Omega(\cdot)$ (Ω -grande)
- Cota exacta (superior e inferior): $\Theta(\cdot)$

Notación o -pequeña

$f(n) = o(g(n))$ ssi $g(n)$ es **significativamente mayor** a $f(n)$

Es decir,

$$\frac{f(n)}{g(n)} \rightarrow 0 \quad \text{cuando} \quad n \rightarrow \infty$$

i.e. para todo $\epsilon > 0$, existe entero n_0 tal que para todo $n \geq n_0$:

$$\frac{f(n)}{g(n)} < \epsilon$$

Notación ω -pequeña

$f(n) = \omega(g(n))$ ssi $g(n)$ es **significativamente menor** a $f(n)$

i.e. $g(n) = o(f(n))$

Es decir,

$$\frac{f(n)}{g(n)} \rightarrow \infty \quad \text{cuando} \quad n \rightarrow \infty$$

Notación O -grande (cota superior)

$f(n) = O(g(n))$ ssi a partir de cierto momento un múltiplo de $|g(n)|$ **acota** a $|f(n)|$ **por arriba**

Es decir,

existe una constante C y un entero n_0 tal que para todo $n \geq n_0$:

$$|f(n)| \leq C |g(n)|$$

Notación Ω -grande (cota inferior)

$f(n) = \Omega(g(n))$ ssi a partir de cierto momento un múltiplo de $|g(n)|$ **acota** a $|f(n)|$ **por abajo**

Es decir,

existe una constante C y un entero n_0 tal que para todo $n \geq n_0$:

$$|f(n)| \geq C |g(n)|$$

$$f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$$

Notación Θ (cota exacta)

$f(n) = \Theta(g(n))$ ssi

- $f(n) = O(g(n))$
- $f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$

Cota asintótica exacta

Búsqueda lineal

Input: arreglo $A[1 \dots n]$ con n elementos y un elemento x

Output: índice i tal que $A[i] = x$ o el valor NIL

```
1 Linear-Search(array A, int x)
2   for i = 1 to A.length do
3     if A[i] == x
4       return i
5   return nil
```

Tiempo en peor caso: $\Theta(n)$ cuando x no está en A ó $A[n] = x$

Tiempo en caso promedio: $\Theta(n)$

$$\frac{1}{n+1} [\Theta(n) + \sum_{i=1}^n \Theta(i)] = \frac{1}{n+1} \Theta(n + \frac{n(n+1)}{2}) = \Theta(n)$$

Búsqueda binaria

Si el arreglo A está ordenado (de forma creciente o decreciente), podemos hacer una búsqueda sobre A de forma más eficiente

La idea es comparar el elemento x a buscar con el elemento z guardado en la **mitad del arreglo**, y **descartar la mitad inferior o superior** cuando x sea mayor o menor a z

El procedimiento se repite hasta encontrar el elemento o descartar todos los elementos del arreglo

Búsqueda binaria: Ejemplo

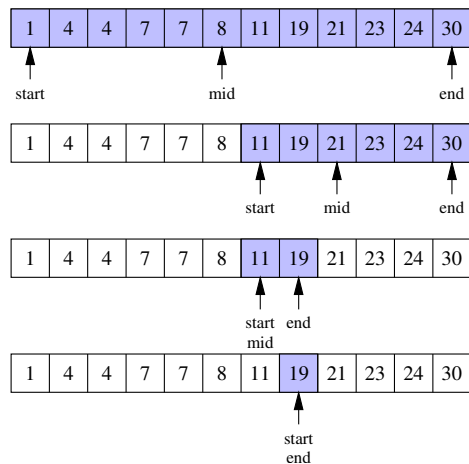


Imagen de <https://puzzle.ics.hut.fi/ICS-A1120/2015/notes/round-efficiency-binarysearch.html>

Búsqueda exitosa del elemento $x = 19$ en un arreglo con 12 elementos:
se realizan 4 comparaciones de x con el elemento mid

Búsqueda binaria: pseudocódigo

Input: arreglo $A[1 \dots n]$ con n elementos **ordenados** y un elemento x

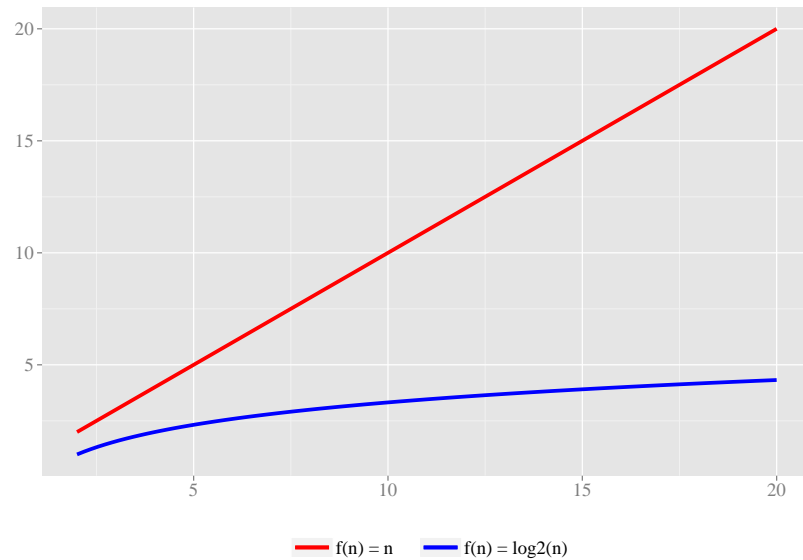
Output: índice i tal que $A[i] = x$ o el valor NIL

```
1 Binary-Search(array A, int x)
2   start = 1
3   end = A.length
4   while start < end do
5     mid = (start + end) / 2           % división entera
6     if A[mid] == x
7       return mid
8     else if A[mid] < x
9       start = mid + 1               % x no está en A[start...mid]
10    else
11      end = mid - 1                 % x no está en A[mid...end]
12  return A[start] == x ? start : nil
```

Tiempo en peor caso: $\Theta(\log n)$

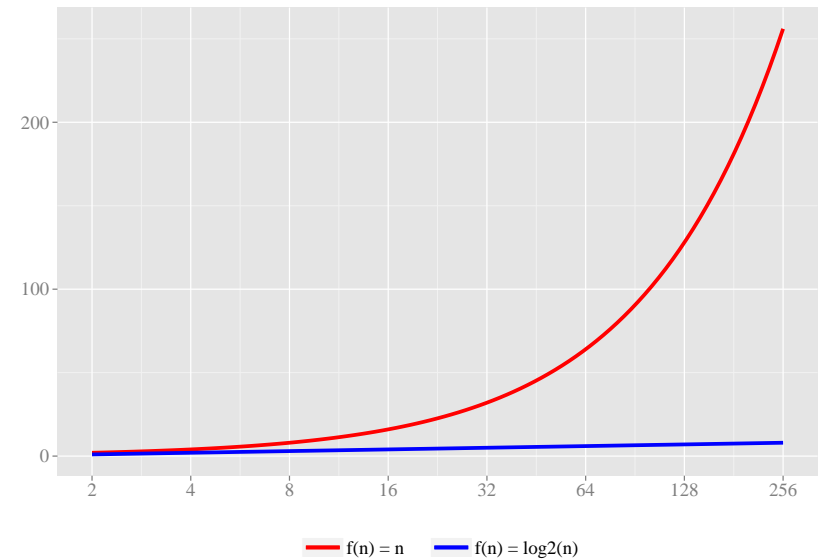
(en cada iteración se descarta la mitad de los elementos restantes)

Tiempo: n vs. $\log(n)$



© 2018 Blai Bonet

Tiempo: n vs. $\log(n)$



© 2018 Blai Bonet

Ordenamiento por inserción

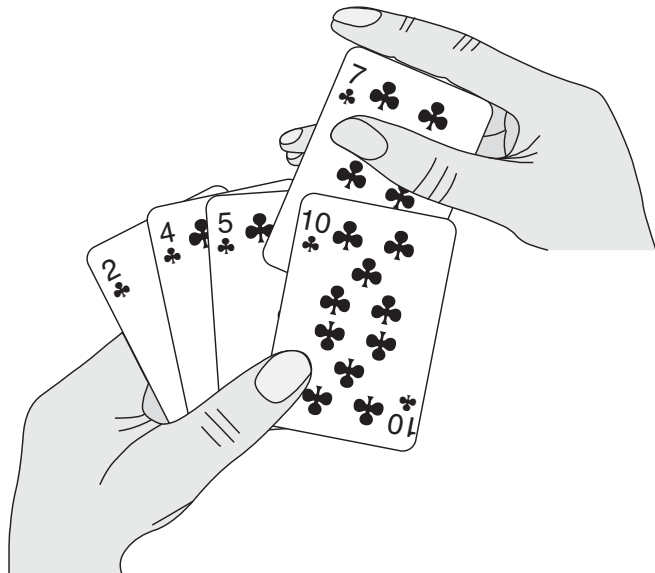


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2018 Blai Bonet

Ordenamiento por inserción

Algoritmo sencillo para ordenar elementos

Método similar al que utiliza la gente para ordenar cartas:

- comienza con un mazo vacío en la mano izquierda y las cartas a ordenar sobre la mesa
- se recoge una carta de la mesa y se inserta en el mazo en la **posición correcta**
- para conseguir la posición correcta, la carta se **compara** con las cartas en el mazo desde la primera (la mayor en el mazo) hasta la última (la menor en el mazo) ó hasta encontrar una carta menor
- se repite el procedimiento hasta insertar todas las cartas en el mazo

© 2018 Blai Bonet

Ordenamiento “in-place” por inserción

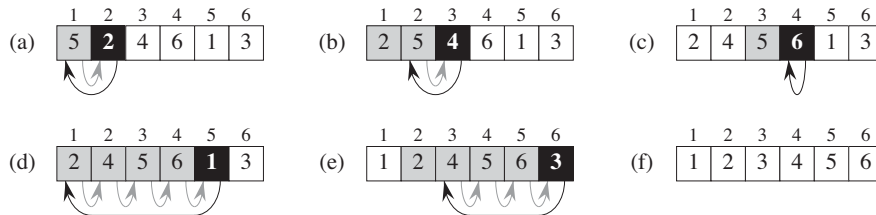


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2018 Blai Bonet

Ordenamiento por inserción

Pseudocódigo de ordenamiento por inserción del arreglo A . El ordenamiento se hace “in place”: los elementos son reordenados dentro del mismo arreglo

Input: arreglo $A[p \dots r]$ con $n = r - p + 1$ elementos

Output: arreglo A con elementos reordenados de menor a mayor

```

1 Insertion-Sort(array A, int p, int r)
2   for j = p + 1 to r do
3     key = A[j]                                % elemento a insertar
4
5     % insertar elemento en la posición correcta
6     i = j - 1
7     while i >= p && A[i] > key do
8       A[i+1] = A[i]
9       i = i - 1
10    A[i+1] = key
  
```

© 2018 Blai Bonet

Correctitud de ordenamiento por inserción

Propiedad del algoritmo:

Al comienzo de cada iteración del lazo, el subarreglo $A[p \dots j - 1]$ consiste de los elementos originalmente en $A[p \dots j - 1]$ pero ordenados de menor a mayor

Propiedad se llama **invariante de lazo**

Si el invariante es cierto, al terminar el lazo (iteración $j = r + 1$), el subarreglo $A[p \dots r]$ está ordenado y por lo tanto el algoritmo es **correcto**

© 2018 Blai Bonet

Invariantes de lazo

Para establecer la certeza de un invariante de lazo, debemos mostrar tres cosas:

Inicialización: el invariante es cierto justo antes de la primera iteración del lazo

Mantenimiento: si el invariante es cierto antes del inicio de una iteración, el invariante sigue siendo cierto después de finalizar la iteración (incluye incremento de variable inductiva)

Terminación: cuando el lazo termina, el invariante nos da una propiedad útil para probar la correctitud del algoritmo

© 2018 Blai Bonet

Correctitud de ordenamiento por inserción

Input: arreglo $A[p \dots r]$ con $n = r - p + 1$ elementos

Output: arreglo A con elementos reordenados de menor a mayor

```
1 Insertion-Sort(array A, int p, int r)
2   for j = p + 1 to r do
3       key = A[j]                % elemento a insertar
4
5       % insertar elemento en la posición correcta
6       i = j - 1
7       while i >= p && A[i] > key do
8           A[i+1] = A[i]
9           i = i - 1
10          A[i+1] = key
```

Correctitud de ordenamiento por inserción

Invariante:

Al comienzo de cada iteración del lazo, el subarreglo $A[p \dots j - 1]$ consiste de los elementos originalmente en $A[p \dots j - 1]$ pero ordenados de menor a mayor

Inicialización: justo antes de la primera iteración, $j = p + 1$. El invariante dice que el subarreglo $A[p \dots j - 1] = A[p \dots p]$ contiene los elementos originalmente en $A[p \dots p]$ y están ordenados de menor a mayor

Claramente es cierto porque el subarreglo contiene un solo elemento

Correctitud de ordenamiento por inserción

Invariante:

Al comienzo de cada iteración del lazo, el subarreglo $A[p \dots j - 1]$ consiste de los elementos originalmente en $A[p \dots j - 1]$ pero ordenados de menor a mayor

Mantenimiento: asuma que estamos por comenzar la j -ésima iteración y que el invariante es cierto

Informalmente, el lazo interno mueve los elementos $A[j - 1], \dots, A[k]$ una posición a la derecha e inserta $A[j]$ en la posición $A[k]$, donde $p \leq k < j$ es único tal que $A[k - 1] \leq A[j] < A[k]$ ó $k = p$ si tal k no existe

Al terminar de ejecutar la asignación en la línea 9, $A[p \dots j]$ contiene los elementos originales en $A[p \dots j]$ de forma ordenada. Por lo tanto, el invariante es cierto después de incrementar j en 1

Correctitud de ordenamiento por inserción

Invariante:

Al comienzo de cada iteración del lazo, el subarreglo $A[p \dots j - 1]$ consiste de los elementos originalmente en $A[p \dots j - 1]$ pero ordenados de menor a mayor

Terminación: el lazo termina cuando $j > r$. Al finalizar la última iteración del lazo, j se incrementa hasta $j = r + 1$ y el invariante sigue siendo cierto por mantenimiento. Por lo tanto, el arreglo $A[p \dots r]$ contiene los elementos originales en A ordenados de forma creciente

Entonces podemos concluir que el **algoritmo es correcto**

Análisis de ordenamiento por inserción

Sea $T(n)$ el tiempo en el peor caso del algoritmo para arreglos de tamaño n

Claramente el lazo externo realiza $n - 1$ iteraciones. Cada lazo interno puede realizar $j - p$ iteraciones ya que la variable i comienza en $j - 1$ y el lazo termina cuando $i = p$

Por lo tanto,

$$T(n) \leq \sum_{j=p+1}^r O(j-p) = \sum_{j=1}^{r-p} O(j) = O\left(\frac{n(n-1)}{2}\right) = O(n^2)$$

donde $n = r - p + 1$ es el número de elementos en el arreglo

Análisis de ordenamiento por inserción

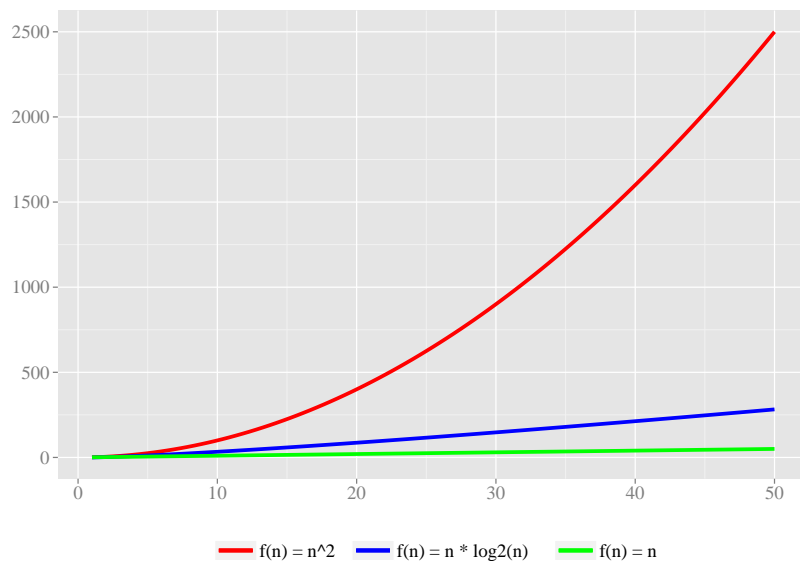
Sea $T(n)$ el tiempo en el peor caso del algoritmo para arreglos de tamaño n

Por otro lado, no es difícil ver que si el arreglo está inicialmente ordenado de mayor a menor, cada lazo interno toma $j - p$ iteraciones:

$$T(n) \geq \sum_{j=p+1}^r \Omega(j-p) = \sum_{j=1}^{r-p} \Omega(j) = \Omega\left(\frac{n(n-1)}{2}\right) = \Omega(n^2)$$

Por lo tanto, $T(n) = \Theta(n^2)$

Tiempo: n vs. $n \log n$ vs. n^2



Resumen

- Algoritmo, modelo computacional y complejidad en tiempo y espacio
- Crecimiento de funciones y notación asintótica
- Búsqueda lineal y binaria sobre un arreglo
- Ordenamiento por inserción

Ejercicios (1 de 3)

1. Haga una búsqueda binaria de $x = 6$ en el arreglo $\langle 1, 4, 4, 7, 7, 8, 11, 19, 21, 23, 24, 30 \rangle$
2. Demuestre la correctitud del algoritmo de búsqueda binaria. Defina un invariante y demuéstrelo. Puede separar los casos cuando x está en el arreglo y cuando no
3. (2.1-1) Ejecute **Insertion-Sort** sobre el arreglo $\langle 31, 41, 59, 26, 41, 58 \rangle$
4. (2.1-2) Modifique **Insertion-Sort** para que ordene de forma decreciente en lugar de creciente

Ejercicios (2 de 3)

5. (2.2-2) Considere un algoritmo de ordenamiento para el arreglo $A[1 \dots n]$ que primero busca el menor elemento en $A[1 \dots n]$ y lo intercambia con $A[1]$. Luego busca el menor elemento en $A[2 \dots n]$ y lo intercambia con $A[2]$, y repite el procesor $n - 1$ veces

Dicho algoritmo es conocido como **Selection-Sort**. Escriba el pseudocódigo de **Selection-Sort**

- a. ¿Cuál es el invariante de lazo que debe utilizarse para probar la correctitud del algoritmo?
- b. ¿Por qué solo hace falta repetir el lazo $n - 1$ veces y no n veces?
- c. ¿Cuál es la complejidad en tiempo de **Selection-Sort** en el mejor y peor caso?

Ejercicios (3 de 3)

6. (2.1-4) Considere el problema de sumar dos enteros de n bits que se encuentran almacenados en dos arreglos A y B de n -elementos. La suma de los dos enteros debe ser almacenada en un arreglo C de $n + 1$ elementos. Diseñe un algoritmo que compute la suma de los números almacenados en A y B , y que guarde el resultado en el arreglo C

7. (2-4) Inversiones

Considere el arreglo $A[1 \dots n]$ con n elementos **distintos**. Si $i < j$ y $A[i] > A[j]$, el par (i, j) es llamado una **inversión** en A

- a. Diga cuales son las 5 inversiones en el arreglo $\langle 2, 3, 8, 6, 1 \rangle$
- b. ¿Cuál arreglo sobre los enteros $\{1, \dots, n\}$ tiene el mayor número de inversiones? ¿Cuántas tiene?
- c. ¿Cuál es la relación entre el número de inversiones en A y el tiempo de corrida de **Insertion-sort** sobre A ?