Análisis y Diseño de Algoritmos

Introducción: El Rol de los Algoritmos en Computación

DR. JESÚS A. GONZÁLEZ BERNAL
CIENCIAS COMPUTACIONALES
INAOE

Temario

1. Introducción
2. Notación Asintótica
3. Recurrencias
4. Ordenamiento
5. Programación Dinámica
6. Algoritmos Voraces
7. Algoritmos Elementales para Grafos
8. Árboles de Expansión Mínima
9. Ruta más Corta
10. Todos los Pares de Ruta más Corta
11. Máximo Flujo
12. Algoritmos Paralelos
13. Teoría de “NP-Completeness”

Evaluación

- 3 Exámenes, 20% cada uno
- 1 Proyecto (investigación y reporte), 25%
- Tareas y presentación, 15%

- Página del curso:
  - http://ccc.inaoep.mx/~jagonzalez/ADA.html

- Contacto
  - jagonzalez@inaoep.mx
  - Oficina: 8303

Algoritmos

- Informalmente
  - Cualquier procedimiento computacional bien definido
    - Valores de entrada
    - Valores de salida
  - Secuencia computacional de pasos que transforman una entrada en una salida
  - Herramienta para resolver un problema computacional bien especificado

Problema de Ordenamiento

- Entrada: secuencia de números $(a_1, a_2, \ldots, a_n)$
- Salida: permutación (reordenamiento) $(a'_1, a'_2, \ldots, a'_n)$ de la secuencia de entrada tal que $a'_1 <= a'_2 <= \ldots <= a'_n$.
- Para (31, 41, 59, 26, 41, 58), la salida es
  - (26, 31, 41, 41, 58, 59)
- A la secuencia de entrada se le llama Instancia
- Una instancia del problema es una entrada
  - Satisface las restricciones del problema
  - Necesaria para calcular la solución al problema

Problema de Ordenamiento

- Ordenamiento es una operación fundamental en computación
  - Utilizado como paso intermedio
  - Hay muchos algoritmos de ordenamiento buenos
  - ¿Cuál es el mejor para una aplicación dada?
    - Depende de:
      - ¿Número de elementos a ordenar?
      - ¿Algunos elementos ya están ordenados?
      - ¿Restricciones en los valores de los elementos?
      - ¿Qué tipo de almacenamiento usamos?
        - » Memoria principal, disco, cintas
- Vamos a trabajar con varios algoritmos de ordenamiento

Algoritmo Correcto

- Se dice que un algoritmo es correcto si para toda instancia de entrada, se detiene al obtener la salida correcta
- Decimos que un algoritmo correcto resuelve el problema computacional dado
- Un algoritmo no-correcto puede no detenerse para algunas instancias de entrada o detenerse con una respuesta incorrecta
  - Los algoritmos incorrectos pueden ser útiles, si podemos controlar su tasa de error
  - Sin embargo, nos concentraremos sólo en algoritmos correctos

Especificación de un Algoritmo

- Inglés, español
- Como programa computacional
- Pseudocódigo
- Sólo una restricción
  - La especificación debe proveer una descripción precisa del procedimiento computacional a seguir

¿Qué Clases de Problemas Resuelven los Algoritmos?

- Muchos más que el ordenamiento
  - El proyecto del Genoma Humano, identificar 100,000 genes en el ADN humano, la secuencia de 3 billones de pares de bases químicas que componen el ADN…
  - Algoritmos de búsqueda para internet, con grandes volúmenes de datos
  - Comercio electrónico (criptografía, firmas digitales)
  - Industrias

Características Comunes de Algoritmos

- Muchas soluciones candidatas, la mayoría no nos interesan, encontrar la que queremos puede ser difícil
- Tienen aplicaciones prácticas, ruta más corta

Estructuras de Datos

- Una estructura de datos es una manera de almacenar y organizar datos para facilitar su acceso y modificación
- Una estructura de datos no trabaja bien para todo propósito, tienen ventajas y limitaciones

Técnica

- Puede que el algoritmo que necesitamos no esté publicado
- Necesitamos una técnica para diseñar y analizar algoritmos para que podamos desarrollar nuevos
  - Demostrar que producen la respuesta correcta
  - Entender su eficiencia

Problemas Duros

- Hay algunos problemas para los que no se conoce una solución eficiente
  - Algoritmos NP-completos
  - No se ha encontrado un algoritmo eficiente para un problema NP-completo
    - Tampoco se sabe si existen (o no) algoritmos eficientes para problemas NP-completos
  - Propiedad de algoritmos NP-completos
    - Si existe un algoritmo eficiente para alguno de ellos, entonces existen algoritmos eficientes para todos ellos
  - Varios problemas NP-completos son similares (no idénticos) a problemas para los que sabemos que hay algoritmos eficientes
    - Cambio pequeño en la definición del problema cause un gran cambio en la eficiencia del mejor algoritmo conocido

Problemas NP-Completos

- Es bueno saber sobre problemas NP-completos
  - Aparecen frecuentemente en aplicaciones reales
  - Podemos invertir mucho tiempo en ellos
  - Si demostramos que el algoritmo es NP-completo
    - Invertir tiempo en desarrollar un algoritmo que obtenga una buena respuesta (no la mejor posible)

Tarea

- Traer para la siguiente clase 2 ejemplos de “Problemas” NP-Completos

Algoritmos como una Tecnología

- ¿Qué pasaría si las computadoras fueran infinitamente rápidas y la memoria fuera gratis?
- Pero como no son infinitamente rápidas y la memoria no es gratuita
  - El tiempo de cómputo y el espacio en memoria son recursos limitados
    - Hay que utilizarlos inteligentemente
    - Algoritmos eficientes en tiempo y espacio

Eficiencia

- Algoritmos que resuelven el mismo problema frecuentemente tienen una eficiencia diferente
  - Más significativas que las diferencias por HW o SW
- Algoritmos de ordenamiento
  - **InsertionSort** $c_1 n^2$ para ordenar n elementos, c_1 es constante no dependiente de n
  - **MergeSort** $c_2 n \lg n$, donde $\lg n$ es $\log_2 n$ y c_2 es otra constante que no depende de n
  - $c_1 < c_2$
    - factores constantes no tan importantes como n
    - n mucho mayor a $\lg n$
    - InsertionSort más rápido que MergeSort para entradas pequeñas
    - Hay un punto (tamaño de n) en el que MergeSort ya es más rápido que InsertionSort

Ejemplo de Eficiencia

- Computadora A más rápida que Computadora B
  - A, 100 millones de inst. x seg. → 10^8 ins/sec
  - B, 1 millón de inst. x seg. → 10^6 ins/sec
- Implementamos InsertionSort en A
  - $2n^2$ ($c_1 = 2$)
- Implementamos MergeSort en B
  - $50n\lg n$ ($c_2 = 50$)
- Ordenar 1,000,000 de números ($n = 10^6$)

Ejemplo de Eficiencia

$$A \rightarrow \frac{2(10^6)^2 inst}{10^8 inst/\text{sec}} = 20{,}000 \text{ seconds} \approx 5.56 \text{ hours},$$

$$B \rightarrow \frac{50 \cdot 10^6 \lg 10^6 inst}{10^6 inst/\text{sec}} \approx 1{,}000 \text{ seconds} \approx 16.67 \text{ minutes}.$$

- Con 10,000,000 de números:
  - B (20 min) es 20 veces más rápida que A (2.3 días) para ordenar los números
    - Ventaja de la eficiencia de MergeSort

Análisis de Algoritmos

- Predecir los recursos que requiere el algoritmo
  - Memoria, tiempo de cómputo
- Modelo de la tecnología de implementación
  - 1 procesador
  - Modelo RAM
    - Una instrucción tras otra, no operaciones concurrentes
- Requerimos herramientas matemáticas

Análisis de InsertionSort

- Tiempo de InsertionSort depende de la entrada
  - Tamaño de la entrada
  - ¿Qué tan ordenada ya está la entrada?
- Tamaño de la entrada (*input size*)
  - Número de elementos en la entrada
    - Vector, número de elementos
    - Grafos, número de vértices y número de arcos

Análisis de InsertionSort

- Tiempo de ejecución (running time)
  - Número de operaciones primitivas (pasos) ejecutadas
- Definir la noción de "paso" de manera independiente de la máquina
  - Tiempo constante para ejecutar cada línea de pseudocódigo
    - Suma, multiplicación, división, comparación, etc.
    - Toman diferente tiempo pero asumimos que es el mismo tiempo y es constante

Análisis de InsertionSort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted		
▷ sequence $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

Análisis Inicial de InsertionSort

- Para cada $j = 2, 3, \ldots, n$, $n = length[A]$
  - t_j es el número de ejecuciones del ciclo while de la línea 5 para el valor de j
- Los comentarios no se ejecutan
- Tiempo de ejecución es la suma de los tiempos de cada línea

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \; .$$

Análisis Inicial de InsertionSort

- Mejor caso
  - Arreglo previamente ordenado
  - Se puede expresar como $an + b$, una función lineal de n.

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \; .
\end{aligned}
$$

Análisis Inicial de InsertionSort

- Peor caso
  - El arreglo tiene un orden decreciente
  - Comparamos cada elemento $A[j]$ con cada elemento del subarreglo $A[1..j\text{-}1]$ y $t_j = j$ para $j = 2, 3, \ldots, n$; donde:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

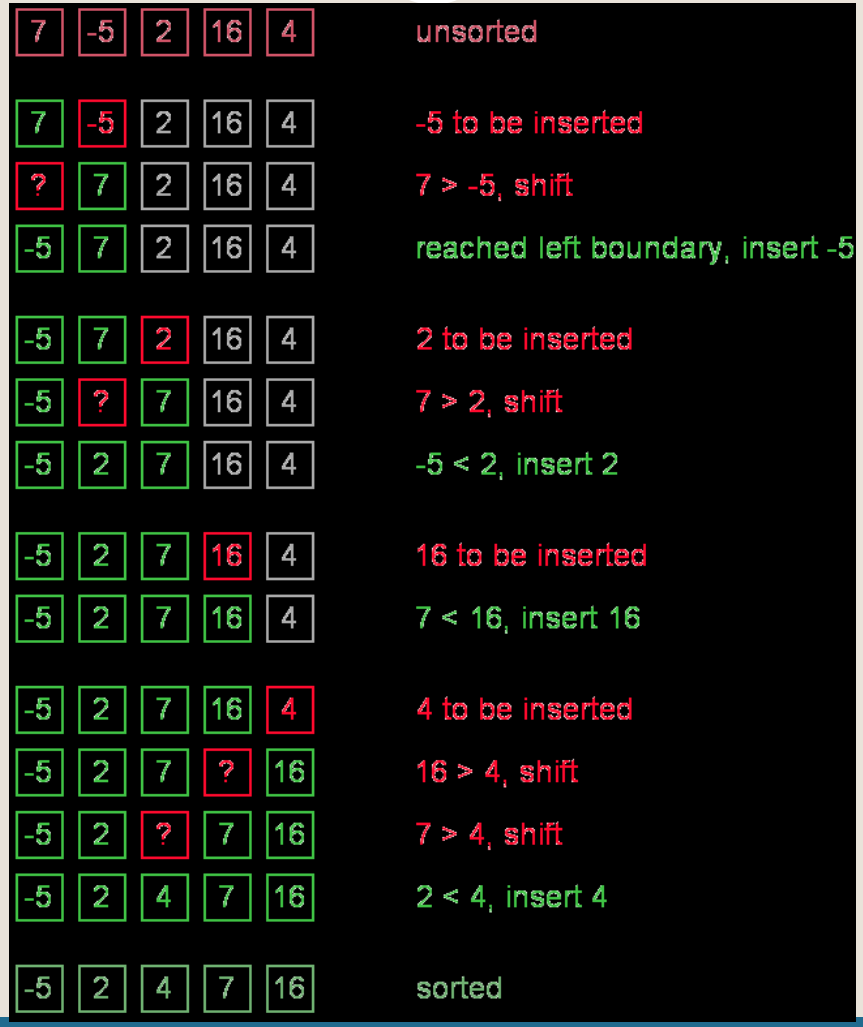
$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Análisis Inicial de InsertionSort

- Peor caso
  - Se puede expresar como $an^2 + bn + c$, es una función cuadrática

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) \; .
\end{aligned}
$$

Ejemplo de InsertionSort



Análisis del Peor Caso y Caso Promedio

- Normalmente calculamos sólo el peor caso
  - El tiempo más largo de ejecución para cualquier entrada de tamaño *n*
- El peor caso del tiempo de ejecución es la frontera más alta
  - Garantizamos que el alg. nunca tardará más que eso
- Para algunos algoritmos el peor caso ocurre muy frecuentemente (i.e. búsqueda en BD)
- El caso promedio es frecuentemente tan malo como el peor caso

Orden de Crecimiento

- Simplificamos el análisis de algoritmos
  - Utilizamos constantes para representar los costos de líneas de código
- Otra simplificación (abstracción)
  - Nos interesa sólo la tasa de crecimiento u órden de crecimiento
  - Tomamos en cuenta sólo el término mayor de la fórmula (p.ej. an^2)
  - También ignoramos los coeficientes (mayores) constantes
    - Porque son menos significativos que la tasa de crecimiento

Orden de Crecimiento

- Peor caso de InsertionSort: Peor caso de $\Theta(n^2)$
  - Theta de *n*-cuadrada
- Consideramos que un algoritmo es más eficiente que otro si el tiempo de ejecución de su peor-caso tiene un orden de crecimiento menor
  - Puede haber un error para entradas pequeñas pero no para entradas grandes

Diseñando Algoritmos

- Hay muchas maneras de diseñar algoritmos
  - InsertionSort
    - Método incremental
  - Otros métodos como
    - Divide y conquista

Método de Divide y Conquista

- Muchos algoritmos son recursivos
  - Se llaman a sí mismos una o más veces para resolver subproblemas muy parecidos
  - Siguen un método divide-y-conquista
    - Dividen el problema en subproblemas similares pero más pequeños
    - Resuelve los problemas recursivamente
    - Combina las soluciones para crear la solución al problema original

Método de Divide y Conquista

- 3 pasos en cada nivel de recursión
  - Divide el problema en subproblemas
  - Conquista los problemas resolviéndolos recursivamente, si el problema es trivial (suficientemente pequeño) lo resuelve de manera directa
  - Combina las soluciones para obtener la solución al problema original

MergeSort

- MergeSort sigue el paradigma divide-y-conquista
  - Divide la secuencia de n-elementos a ordenar en 2 subsecuencias de $n/2$ elementos cada una
  - Ordena las 2 subsecuencias recursivamente utilizando MergeSort
  - Combina: intercala las dos subsecuencias ordenadas para producir la respuesta ordenada

MergeSort

Inicialmente:

A es el arreglo de entrada

p =1

r = *length*[*A*]

```
MERGE-SORT(A, p, r)
1  if p < r
2      then q ← ⌊(p + r)/2⌋
3           MERGE-SORT(A, p, q)
4           MERGE-SORT(A, q + 1, r)
5           MERGE(A, p, q, r)
```

MergeSort


```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7        do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

Análisis de Algoritmos Divide y Conquista

- Algoritmo recursivo
  - Su tiempo de ejecución descrito con una ecuación de recurrencia ó recurrencia
    - Describe el tiempo de ejecución de un problema de tamaño *n* en términos del tiempo de ejecución de entradas más pequeñas
  - Herramientas matemáticas para resolver recurrencias

Recurrencias para Algoritmos Divide y Conquista

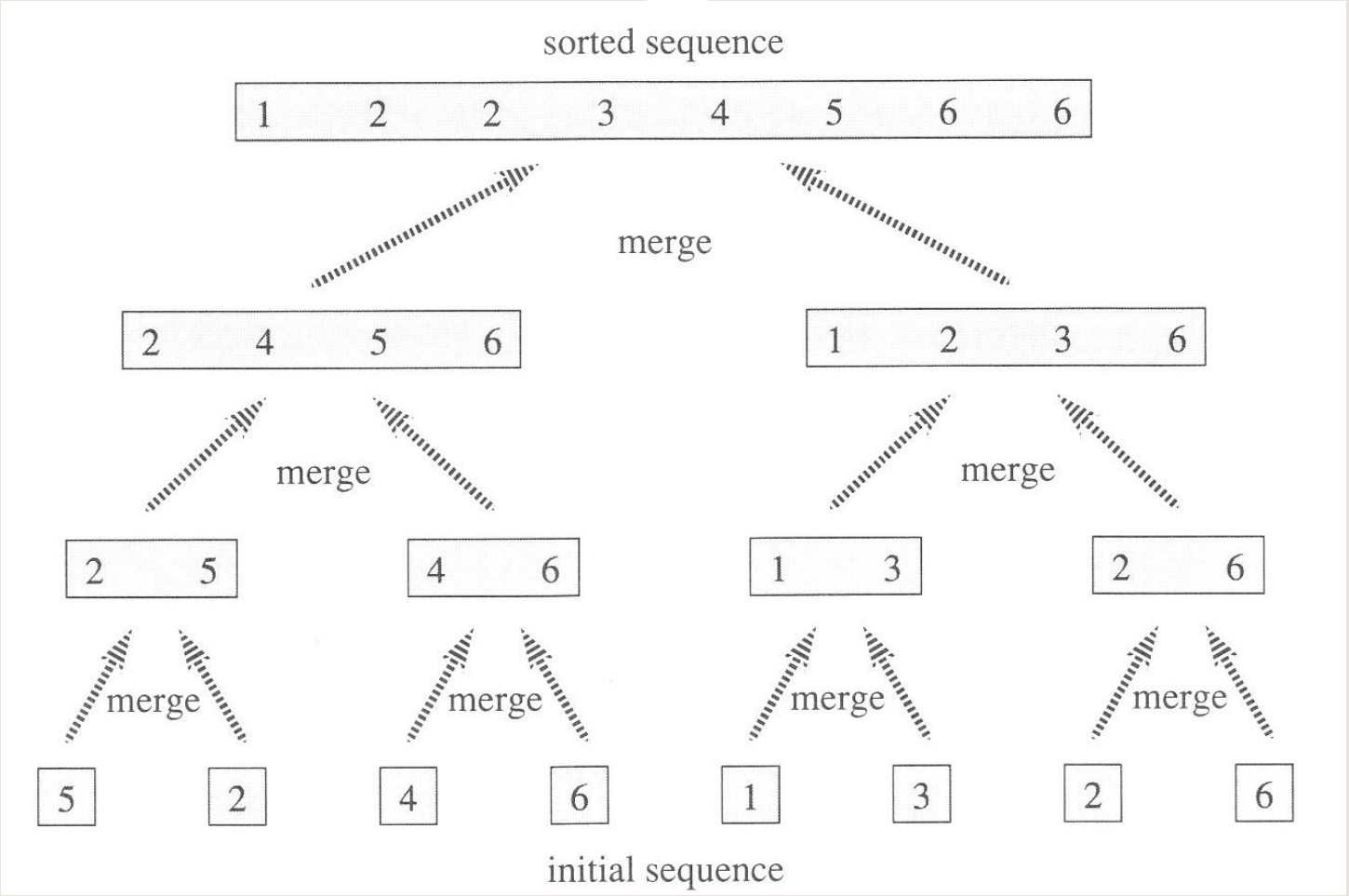
- Basadas en los 3 pasos del paradigma
  - $T(n)$ es el tiempo de ejecución del problema de tamaño n
  - Si el problema es suficientemente pequeño (n menor a una constante c), toma tiempo constante: $\Theta(1)$

Recurrencias para Algoritmos Divide y Conquista

- Al dividir el problema en a subproblemas
  - Cada subproblema de tamaño $1/b$ del tamaño del original
  - $D(n)$ tiempo en dividir el problema
  - $C(n)$ tiempo en combinar las soluciones
  - Recurrencia obtenida:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

Análisis de MergeSort



Análisis de MergeSort

- Asume problema tamaño potencia de 2
  - Cada división de problema genera 2 subsecuencias de tamaño $n/2$
- Tiempo de ejecución en el peor caso para MergeSort con n números
  - MergeSort con un sólo número → Tiempo Constante
  - Para n > 1 dividimos el problema:
    - Divide → Calcula la mitad del subarreglo, toma tiempo constante, $D(n)=\Theta(1)$
    - Conquista → Resuelve los 2 subproblemas recursivamente con $2T(n/2)$ de tiempo de ejecución
    - Combina → $C(n)= \Theta(n)$

Análisis de MergeSort

- Al sumar las funciones $D(n)$ y $C(n)$ para el análisis
  - Sumamos funciones $\Theta(n)$ y $\Theta(1)$, tenemos una función lineal de n, $\Theta(n)$
- Al añadir el término $2T(n/2)$, da el peor caso de MergeSort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

  - $T(n)$ es $\Theta(n \lg n)$, $\lg n$ es $\log_2 n$
  - Para entradas suficientemente grandes, MergeSort con $\Theta(n \lg n)$ es mejor que InsertionSort con $\Theta(n^2)$