Repaso I1

Clase 12

IIC 2133 - Sección 3

Prof. Eduardo Bustos

Sumario

SelectionSort

Lema: *seleccionar de forma ordenada*

Funcionamiento a alto nivel

1. Seleccionar menor elemento
2. Ubicarlo como último elemento en zona *ordenada*
3. Repetir hasta seleccionar **todos** los elementos

Desempeño en casos

- No distingue secuencias por su orden a priori
- Mismo número de comparaciones en todas las secuencias

Complejidad en la secuencia $A[0 \dots n-1]$

- Versión original: tiempo $\mathcal{O}(n^2)$ y memoria $\mathcal{O}(n)$
- Versión *in place*: tiempo $\mathcal{O}(n^2)$ y memoria $\mathcal{O}(1)$

SelectionSort

input : Secuencia $A[0 \dots n-1]$, largo $n \geq 2$
output: $\varnothing$

`SelectionSort` (A, n)**:**

for $i = 0 \dots n-2$ **:**
   $min \leftarrow i$
   for $j = i+1 \dots n-1$ **:**
      if $A[j] < A[min]$ **:**
         $min \leftarrow j$
   $A[i] \leftrightharpoons A[min]$

Versión *in place* en tiempo $\mathcal{O}(n^2)$

InsertionSort

Lema: *insertar de forma ordenada*

Funcionamiento a alto nivel

1. Seleccionar el primer elemento no revisado
2. Moverlo hasta su posición correcta
3. Repetir hasta ubicar **todos** los elementos

Desempeño en casos

- Como revisa si el elemento está bien insertado...
- ...*se da cuenta* cuando un elemento está ordenado

Complejidad en la secuencia $A[0 \dots n-1]$

- Mejor caso: secuencia ordenada
  - Versión *in place*: tiempo $\mathcal{O}(n)$ y memoria $\mathcal{O}(1)$
- Todos los demás casos:
  - Versión *in place*: tiempo $\mathcal{O}(n^2)$ y memoria $\mathcal{O}(1)$

InsertionSort

input : Secuencia $A[0 \dots n-1]$, largo $n \geq 2$

output: $\varnothing$

`InsertionSort` *(A, n)*:

```
for i = 1 ... n − 1 :
    j = i
    while (j > 0) ∧ (A[j] < A[j − 1]) :
        Intercambiar A[j] con A[j − 1]
        j = j − 1
```

Versión *in place*: mejor caso $\mathcal{O}(n)$, e.o.c. $\mathcal{O}(n^2)$

MergeSort

Lema: *mezclar mitades ordenadas*

Funcionamiento a alto nivel

1. Recursivamente dividir secuencia en mitades
2. Caso base: secuencias de largo 1
3. Mezclar ordenadamente subsecuencias ordenadas (`Merge`)

Desempeño

- Cantidad de llamados recursivos **siempre** logarítmica
- Mezcla lineal en todos los casos

Complejidad en la secuencia $A[0 \ldots n-1]$

- Versión *in place*: tiempo $\mathcal{O}(n \log(n))$ y memoria $\mathcal{O}(n)$

MergeSort

input : Secuencia A
output: Secuencia ordenada B

`MergeSort` *(A)*:

- **if** $|A| = 1$ **: return** A
- Dividir A en A_1 y A_2
- $B_1 \leftarrow \texttt{MergeSort}(A_1)$
- $B_2 \leftarrow \texttt{MergeSort}(A_2)$
- $B \leftarrow \texttt{Merge}(B_1, B_2)$
- **return** B

`Merge`(A, B):

- Iniciar C vacía
- **while** $|A| > 0 \wedge |B| > 0$ **:**
  - **if** $A[1] \leq B[1]$ **:**
    - $e \leftarrow \texttt{Extraer}(A[1])$
  - **else:**
    - $e \leftarrow \texttt{Extraer}(B[1])$
  - Insertar e al final de C
- Concatenar C con la secuencia restante
- **return** C

Tiempo $\mathcal{O}(n \log(n))$ y memoria $\mathcal{O}(n)$

QuickSort

Lema: *ordenar pivotes de forma recursiva*

Funcionamiento a alto nivel

1. Recursivamente ordenar un elemento arbitrario (pivote)
2. Caso base: secuencias de largo 0
3. No requiere mezclar: `Partition` ordena elementos

Desempeño

- Depende de la elección del pivote
- De antemano no podemos anticipar el desempeño: probabilístico

Complejidad en la secuencia $A[0 \ldots n-1]$

- Mejor caso y promedio
  - Versión *in place*: tiempo $\mathcal{O}(n \log(n))$ y memoria $\mathcal{O}(1)$
- Peor caso: pivote *mágicamente malo* siempre
  - Versión *in place*: tiempo $\mathcal{O}(n^2)$ y memoria $\mathcal{O}(1)$

QuickSort

input : Secuencia $A[0, \ldots, n-1]$, índices i, f

output: $\varnothing$

`QuickSort` (A, i, f)**:**

- **if** $i \leq f$ **:**
  - $p \leftarrow \texttt{Partition}(A, i, f)$
  - $\texttt{Quicksort}(A, i, p-1)$
  - $\texttt{Quicksort}(A, p+1, f)$

`Partition` (A, i, f)**:**

- $x \leftarrow$ índice aleatorio en $\{i, \ldots, f\}$; $\quad p \leftarrow A[x]$
- $A[x] \rightleftarrows A[f]$
- $j \leftarrow i$
- **for** $k = i \ldots f-1$ **:**
  - **if** $A[k] < p$ **:**
    - $A[j] \rightleftarrows A[k]$
    - $j \leftarrow j+1$
- $A[j] \rightleftarrows A[f]$
- **return** j

Caso promedio y mejor caso: tiempo $\mathcal{O}(n \log(n))$ y memoria $\mathcal{O}(1)$

Mejoras en `Quicksort`

- Para sub-secuencias pequeñas (e.g. $n \leq 20$) podemos usar `InsertionSort`
  - A pesar de no tener una complejidad mejor
  - Eso es solo cuando hablamos asintóticamente
  - En la práctica, en instancias pequeñas tiene mejor desempeño
  - No olvidar que `Quicksort` es **recursivo**... eso cuesta recursos
- Usar la mediana de tres elementos como pivote
  - *Informar* la elección de pivote
  - Dado A, escogemos 3 elementos $A[k_1], A[k_2], A[k_3]$
  - En $\mathcal{O}(1)$ encontramos la mediana entre ellos
- Particionar la secuencia en 3 sub-secuencias: menores, iguales y mayores
  - Mejora para caso con datos repetidos
  - Evita que `Partition` particione innecesariamente sub-secuencias en que todos los valores son iguales

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Heap Sort	?	?	?	?

Estrategias algorítmicas

Además de estudiar algoritmos iterativos, vimos dos ejemplos recursivos de la estrategia **dividir para conquistar**

A saber,

- `MergeSort`
- `QuickSort`

Dividir para conquistar

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo
2. Resolver **recursivamente** cada sub-problema
3. Encontrar solución al problema original **combinando** las soluciones a los sub-problemas

Los sub-problemas son instancias más pequeñas del problema a resolver

Un ejemplo adicional: Búsqueda binaria

El algoritmo de **búsqueda binaria** está basado en la estrategia dividir para conquistar

`BSearch` (A, x, i, f)**:**

    if $f < i$ **: return** -1

    $m \leftarrow \left\lfloor \frac{i+f}{2} \right\rfloor$

    if $A[m] = x$ **: return** m

    if $A[m] > x$ **:**

        return `BSearch` $(A, x, i, m-1)$

    return `BSearch` $(A, x, m+1, f)$

Recordar: ciertos algoritmos D.P.C. no resuelven todos los subproblemas

Sumario

Diccionarios

Definición

Un **diccionario** es una estructura de datos con las siguientes operaciones

- **Asociar** un valor a una llave
- **Actualizar** el valor asociado a una llave
- **Obtener** el valor asociado a una llave
- En ciertos casos, **eliminar** de la estructura una asociación llave-valor

Objetivo central: búsqueda eficiente

Diccionarios: dos enfoques

Vimos dos instancias de diccionarios

1. Árboles de búsqueda
   - Binarios AVL
   - 2-3
   - Binarios rojo-negro
2. Tablas de Hash

En esta interrogación evaluaremos hasta árboles.
No se evaluará tablas de hash!

Árboles de búsqueda

Tres tipos estudiados: binarios AVL y rojo-negro, y árboles 2-3

Aspectos esenciales

- Los tres tipos tienen la **propiedad de búsqueda**: valores ordenados en

  $$\text{hijo izquierdo} < \text{padre} < \text{hijo derecho}$$

- Cada tipo tiene una noción de **balance** dada por sus operaciones
  - AVL cuida la altura de sus hijos recursivamente
  - 2-3 mantiene balance a través de la mantención de hojas en un mismo nivel
  - Rojo-negro cuida la cantidad de nodos negros hacia las hojas

Importancia del balance: mantener el árbol con profundidad logarítmica

Árboles de búsqueda

Operaciones

- Búsqueda gracias a la propiedad de búsqueda
- Inserción
  - AVL: involucra posibles rotaciones
  - 2-3: involucra posibles splits
  - Rojo-negro: involucra posibles rotaciones y cambios de color
- Eliminación: en general compleja y recursiva

Todas estas son operaciones $\mathcal{O}(\log(n))$ cuando $h \in \mathcal{O}(\log(n))$

Las operaciones se benefician del balance

Árboles de búsqueda

Orientaciones para el estudio

- Comprender la estrategia de balance de cada tipo
- Construir árboles pequeños con inserción de llaves suscesivas
- Definir pequeñas secuencias de inserción que generan árboles más/menos desbalanceados
- Comparar el desempeño de los árboles en posibles situaciones prácticas. ¿Cuál se puede portar mejor?

Sumario

Interrogación 1

Objetivos a evaluar en la I1

- ☐ Comprender y comparar algoritmos de ordenación clásicos
- ☐ Modificar algoritmos conocidos para resolver problemas
- ☐ Diseñar algoritmos usando técnicas e ideas estudiadas
- ☐ Demostrar correctitud de algoritmos
- ☐ Analizar complejidad de algoritmos
- ☐ Comprender el funcionamiento de EDDs de árboles
- ☐ Comparar estructuras basadas en árboles

Varios objetivos pueden incluirse en cada pregunta

Interrogación 1

Formato de la prueba

- 2 horas de tiempo
- Pool de 4 preguntas para elegir 3
- Cada pregunta incluye un título que describe sus temas
- ¡**SOLO** se entregan 3 preguntas respondidas!

Nota de la I1: promedio de las 3 preguntas entregadas

Interrogación 1

Material adicional

- Pueden usar un formulario/apuntes durante la prueba
- Debe estar escrito a mano (puede ser impreso de tablet)
- Una hoja (por ambos lados)
- Sugerencia: incluyan los pseudocódigos vistos

No se aceptarán diapositivas impresas

Sumario

Ejemplo 1: Aplicación de algoritmos

Ejercicio (I1 P2 - 2022-2)

Un archivo contiene datos de todos los estudiantes que han rendido cursos del DCC desde 1980 hasta la fecha. El formato cada registro en el archivo es

RUT	primer_apellido	segundo_apellido	nombre

y los registros se encuentran ordenados por `RUT`.

Proponga el pseudocódigo de un algoritmo para ordenar los registros alfabéticamente, i.e. según (`primer_apellido`, `segundo_apellido`, `nombre`). Especifique qué estructura básica usará (listas o arreglos). Si p es un registro, puede acceder a sus atributos con `p.primer_apellido`, `p.nombre`, etc. Además, puede asumir que todo algoritmo de ordenación visto en clase puede ordenar respecto a un atributo específico.

Ejemplo 1: Aplicación de algoritmos

Ejemplo 1: Aplicación de algoritmos

input : Arreglo $A[0, \ldots, n-1]$ e índices i, f
output: Lista de pares de índices L

```
FirstLastNameReps (A, i, f):
    L ← lista vacía
    k ← i
    j ← i
    for m = 1 ... f :
        if A[m].primer_apellido = A[k].primer_apellido :
            j ← m
        else:
            if k < j :
                añadir a L el par (k, j)
            k ← m
            j ← m
    return L
```

Ejemplo 1: Aplicación de algoritmos

`FirstLastNameReps` (A, i, f) entrega una lista con los rangos entre los cuales hay repeticiones de primer apellido entre los índices i y f. De forma similar se define la rutina `SecondLastNameReps` que entrega rangos de repetidos de segundo apellido.

Ejemplo 1: Aplicación de algoritmos

input : Arreglo $A[0, \ldots, n-1]$

output: Arreglo ordenado alfabéticamente

```
AlphaSort (A):
    MergeSort(A, 0, n − 1, primer_apellido)        ▷ A según 1º apellido
    F ← FirstLastNameReps(A, 0, n − 1)
    for (k, j) ∈ F :
        MergeSort(A, k, j, segundo_apellido)
        S ← SecondLastNameReps(A, k, j)
        for (s, t) ∈ S :
            MergeSort(A, s, t, nombre)
```

Ejemplo 1: Aplicación de algoritmos

Ejercicio (I1 P2 - 2022-2)

Determine la complejidad de peor caso de su algoritmo en función del número de estudiantes en el archivo.

Ejemplo 1: Aplicación de algoritmos

El peor caso corresponde a una cantidad $\mathcal{O}(n)$ de repetidos en primer y segundo apellido. Luego, el análisis de complejidad puede resumirse en

- Línea 1 en $\mathcal{O}(n\log(n))$
- Línea 2 en $\mathcal{O}(n)$
- **for** de línea 3 se ejecuta $\mathcal{O}(n)$ veces
  - Línea 4 en $\mathcal{O}(n\log(n))$
  - Línea 5 en $\mathcal{O}(n)$
  - **for** de línea 6 se ejecuta $\mathcal{O}(n)$ veces
    - Línea 7 en $\mathcal{O}(n\log(n))$

Con esto, la complejidad sería

$$\mathcal{O}(n\log(n) + n + n \cdot [n\log(n) + n + n \cdot (n\log(n))]) \quad \Rightarrow \quad \mathcal{O}(n^3\log(n))$$

Ejemplo 2: Dividir para conquistar

Ejercicio (I1 P3 - 2022-2)

Dada una secuencia $A[0 \ldots n-1]$ de números enteros, se define un **índice mágico** como un índice $0 \leq i \leq n-1$ tal que $A[i] = i$. Por ejemplo, en la siguiente secuencia

-7	3	2	5	-1	15	7	12	6	9	3
0	1	2	3	4	5	6	7	8	9	10

existen dos índices mágicos: el 2 y el 9.

Dada una secuencia $A[0 \ldots n-1]$ **ordenada**, sin elementos repetidos e implementada como arreglo,

- proponga el pseudocódigo de un algoritmo que retorne un índice mágico en A si existe y que retorne `null` en caso contrario. Su algoritmo debe ser más eficiente que simplemente revisar el arreglo elemento por elemento, i.e. mejor que $\mathcal{O}(n)$.

Ejemplo 2: Dividir para conquistar

Ejemplo 2: Dividir para conquistar

input : Arreglo $A[0, \ldots, n-1]$, índices i, f
output: Índice mágico o `null`

```
Magic (A, i, f):
    if (f − i) = 0 :
        if A[i] = i :
            return i
        return null
    p ← ⌊(f − i)/2⌋
    if A[p] = p :
        return p
    if A[p] > p :
        return Magic(A, i, p − 1)
    return Magic(A, p + 1, f)
```

Sumario

Recomendaciones finales

Para los algoritmos vistos en clase

- Comprender las demostraciones de correctitud
- Replicarlas por su cuenta
- Asegurarse de poder motivar el *¿por qué se plantea esta propiedad para demostrar?*
- Ser capaz de determinar su complejidad y casos

Guías de ejercicios y pautas anteriores

- Hay harto material resuelto en el repo!
- No se aprendan pautas... seleccionen y aprovéchenlas
- Planifiquen su solución antes de verla, y luego consulten la pauta