Repaso I1

Clase 14

IIC 2133 - Sección 2

Prof. Mario Droguett

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

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

```
for i = 0 ... n − 2 :
    min ← i
    for j = i + 1 ... n − 1 :
        if A[j] < A[min] :
            min ← j
    A[i] ⇋ 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** $i \leq f$ **:**
  - $p \leftarrow$ `Partition`(A, i, f)
  - `Quicksort`$(A, i, p-1)$
  - `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

Heaps binarios

Definición

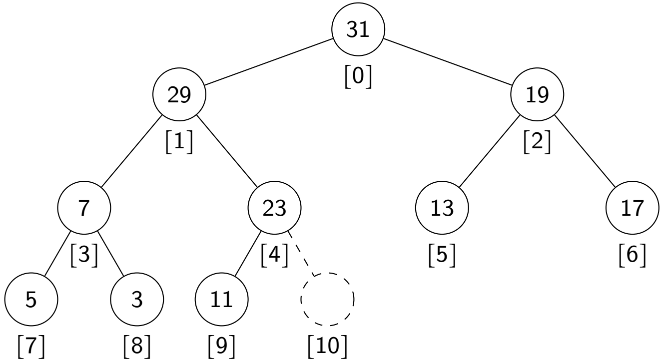
Un **heap binario** H es un árbol binario tal que

- $H.left$ y $H.right$ son heaps binarios
- $H.value > H.left.value$
- $H.value > H.right.value$

A estas condiciones les llamamos **propiedad de heap**

Esta es la definición de un **MAX heap**

Heaps binarios: representación compacta



31	29	19	7	23	13	17	5	3	11	
0	1	2	3	4	5	6	7	8	9	10

Heaps binarios

Aspectos esenciales

- Propiedad de heap

$$\text{descendientes} < \text{nodo} < \text{ancestros}$$

- Se asegura **balance** mediante sus operaciones
  - Inserción al final del arreglo, reubicando (`SiftUp`)
  - Extracción de la raíz, reemplazando con el último y reubicando (`SiftDown`)
  - Actualización de prioridad (`SiftUp`)

Operaciones logarítmicas en la cantidad de nodos

Heaps binarios

Operaciones

- Inserción: agregar un nuevo elemento
  - Se inserta al final del arreglo
  - Se reubica con `SiftUp`
- Extracción: sacar y retornar el más prioritario
  - Se saca el primer elemento y se reemplaza con el último
  - Se reubica la nueva raíz con `SiftDown`
- Actualización: aumentarle la prioridad a un nodo
  - Se reubica el nodo con `SiftUp`

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

> Los heaps no se usan para buscar,
> sino para informar el elemento más prioritario

Heaps binarios

Orientaciones para el estudio

- Comprender la representación compacta de heaps
- Comprender operaciones y su uso para mantener una cola de prioridad
- Considerar casos borde: ¿qué pasa si la prioridad es el orden de llegada? ¿Qué se puede hacer más rápido y cómo?

Incluyan en el formulario el pseudocódigo de los métodos de heaps

Orden lineal

Aspectos esenciales de los algoritmos de orden lineal

- Operan sobre naturales (índices para un arreglo)
- Counting Sort permite ordenar naturales de un conjunto acotado
- La idea se generaliza a Radix para ordenar palabras cualesquiera
  - Cada símbolo se asocia con un natural
  - Requiere un algoritmo estable como Counting Sort

Estos algoritmos ordenan n datos en tiempo $\mathcal{O}(n)$
si la cantidad de símbolos diferentes es $\mathcal{O}(n)$

Orden lineal

Orientaciones para el estudio

- ☐ Comprender cuándo se pueden ocupar los algoritmos lineales
- ☐ Comprender pseudocódigo de Counting Sort y Radix para potenciales modificaciones

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	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(1)$

Sumario

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
   - B+
2. Tablas de Hash

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

Árboles de búsqueda

Cuatro tipos estudiados: binarios AVL y rojo-negro, y árboles 2-3 y B+

Aspectos esenciales

- Los cuatro 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
  - B+: 2-3 generalizados

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
  - B+: posibles splits y reordenamientos
- 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 y 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 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