

Repaso I1

Clase 12

IIC 2133 - Sección 1

Prof. Diego Arroyuelo

Sumario

Algoritmos de ordenación

Dividir para conquistar

Árboles de búsqueda

Interrogación 1

Un ejemplo de pruebas

Cierre

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: \emptyset

SelectionSort (A, n):

```
1  for  $i = 0 \dots n-2$  :  
2       $min \leftarrow i$   
3      for  $j = i+1 \dots n-1$  :  
4          if  $A[j] < A[min]$  :  
5               $min \leftarrow j$   
6       $A[i] \rightleftharpoons A[min]$ 
```

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: \emptyset

InsertionSort (A, n):

```
1   for  $i = 1 \dots n - 1$  :  
2        $j = i$   
3       while  $(j > 0) \wedge (A[j] < A[j - 1])$  :  
4           Intercambiar  $A[j]$  con  $A[j - 1]$   
5            $j = j - 1$ 
```

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 $A[0 \dots n-1]$

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

MergeSort

input : Secuencia A

output: Secuencia ordenada B

MergeSort (A):

```
1  if  $|A| = 1$  : return  $A$ 
2  Dividir  $A$  en  $A_1$  y  $A_2$ 
3   $B_1 \leftarrow \text{MergeSort}(A_1)$ 
4   $B_2 \leftarrow \text{MergeSort}(A_2)$ 
5   $B \leftarrow \text{Merge}(B_1, B_2)$ 
6  return  $B$ 
```

Merge(A, B):

```
1  Iniciar  $C$  vacía
2  while  $|A| > 0 \wedge |B| > 0$  :
3      if  $A[1] \leq B[1]$  :
4           $e \leftarrow \text{Extraer}(A[1])$ 
5      else:
6           $e \leftarrow \text{Extraer}(B[1])$ 
7      Insertar  $e$  al final de  $C$ 
8  Concatenar  $C$  con la
   secuencia restante
9  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 $A[0 \dots n-1]$

- 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

$A[0, \dots, n-1]$,

índices i, f

output: \emptyset

QuickSort (A, i, f):

```
1  if  $i \leq f$  :  
2       $p \leftarrow \text{Partition}(A, i, f)$   
3      Quicksort( $A, i, p-1$ )  
4      Quicksort( $A, p+1, f$ )
```

Partition (A, i, f):

```
1   $x \leftarrow$  índice aleatorio en  
     $\{i, \dots, f\}$ ;  $p \leftarrow A[x]$   
2   $A[x] \rightleftharpoons A[f]$   
3   $j \leftarrow i$   
4  for  $k = i \dots f-1$  :  
5      if  $A[k] < p$  :  
6           $A[j] \rightleftharpoons A[k]$   
7           $j \leftarrow j+1$   
8   $A[j] \rightleftharpoons A[f]$   
9  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

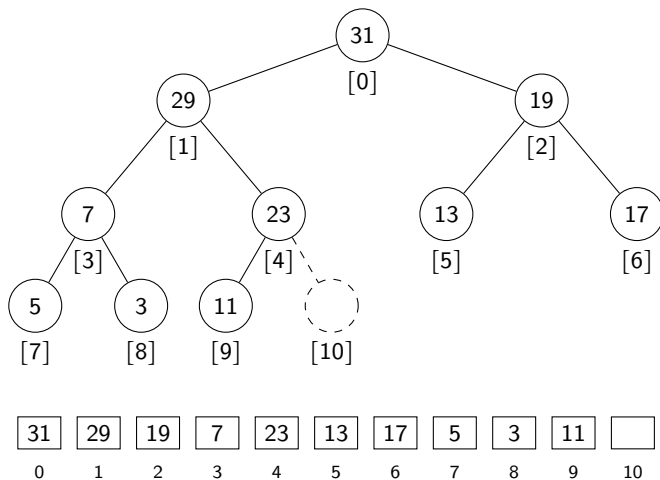
Un **heap binario** H es un árbol binario tal que

- $H.left$ 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



Heaps binarios

Aspectos esenciales

- Propiedad de heap

descendientes < nodo < 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

Algoritmos de ordenación

Dividir para conquistar

Árboles de búsqueda

Interrogación 1

Un ejemplo de pruebas

Cierre

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: \emptyset

SelectionSort (A, n):

```
1  for  $i = 0 \dots n-2$  :  
2       $min \leftarrow i$   
3      for  $j = i+1 \dots n-1$  :  
4          if  $A[j] < A[min]$  :  
5               $min \leftarrow j$   
6       $A[i] \rightleftharpoons A[min]$ 
```

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: \emptyset

InsertionSort (A, n):

```
1  for  $i = 1 \dots n - 1$  :  
2       $j = i$   
3      while  $(j > 0) \wedge (A[j] < A[j - 1])$  :  
4          Intercambiar  $A[j]$  con  $A[j - 1]$   
5           $j = j - 1$ 
```

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 $A[0 \dots n - 1]$

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

MergeSort

input : Secuencia A

output: Secuencia ordenada B

MergeSort (A):

```
1  if  $|A| = 1$  : return  $A$ 
2  Dividir  $A$  en  $A_1$  y  $A_2$ 
3   $B_1 \leftarrow \text{MergeSort}(A_1)$ 
4   $B_2 \leftarrow \text{MergeSort}(A_2)$ 
5   $B \leftarrow \text{Merge}(B_1, B_2)$ 
6  return  $B$ 
```

Merge(A, B):

```
1  Iniciar  $C$  vacía
2  while  $|A| > 0 \wedge |B| > 0$  :
3      if  $A[1] \leq B[1]$  :
4           $e \leftarrow \text{Extraer}(A[1])$ 
5      else:
6           $e \leftarrow \text{Extraer}(B[1])$ 
7      Insertar  $e$  al final de  $C$ 
8  Concatenar  $C$  con la
   secuencia restante
9  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 $A[0 \dots n - 1]$

- 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

$A[0, \dots, n-1]$,

índices i, f

output: \emptyset

QuickSort (A, i, f):

```
1  if  $i \leq f$  :  
2       $p \leftarrow \text{Partition}(A, i, f)$   
3      Quicksort( $A, i, p-1$ )  
4      Quicksort( $A, p+1, f$ )
```

Partition (A, i, f):

```
1   $x \leftarrow$  índice aleatorio en  
     $\{i, \dots, f\}$ ;  $p \leftarrow A[x]$   
2   $A[x] \rightleftharpoons A[f]$   
3   $j \leftarrow i$   
4  for  $k = i \dots f-1$  :  
5      if  $A[k] < p$  :  
6           $A[j] \rightleftharpoons A[k]$   
7           $j \leftarrow j+1$   
8   $A[j] \rightleftharpoons A[f]$   
9  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

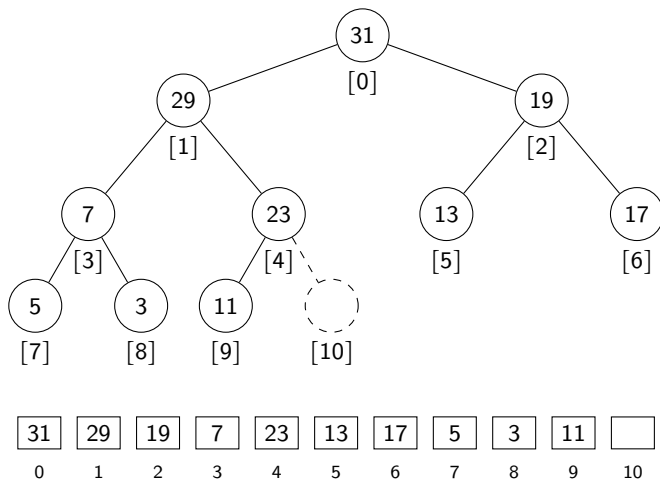
Un **heap binario** H es un árbol binario tal que

- $H.left$ 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



Heaps binarios

Aspectos esenciales

- Propiedad de heap

descendientes < nodo < 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

Algoritmos de ordenación

Dividir para conquistar

Árboles de búsqueda

Interrogación 1

Un ejemplo de pruebas

Cierre

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

hijo izquierdo < padre < 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 sucesivas
- ☐ 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

Algoritmos de ordenación

Dividir para conquistar

Árboles de búsqueda

Interrogación 1

Un ejemplo de pruebas

Cierre

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

Algoritmos de ordenación

Dividir para conquistar

Árboles de búsqueda

Interrogación 1

Un ejemplo de pruebas

Cierre

Ejemplo 2: Dividir para conquistar

Ejercicio (I1 P3 - 2022-2)

Dada una secuencia $A[0 \dots n-1]$ 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 $A[0 \dots n-1]$ 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 $A[0, \dots, n-1]$, índices i, f

output: Índice mágico o null

Magic (A, i, f):

```
1  if  $(f - i) = 0$  :  
2      if  $A[i] = i$  :  
3          return  $i$   
4      return null  
5   $p \leftarrow \lfloor (f - i)/2 \rfloor$   
6  if  $A[p] = p$  :  
7      return  $p$   
8  if  $A[p] > p$  :  
9      return Magic( $A, i, p - 1$ )  
10 return Magic( $A, p + 1, f$ )
```

Sumario

Algoritmos de ordenación

Dividir para conquistar

Árboles de búsqueda

Interrogación 1

Un ejemplo de pruebas

Cierre

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