Repaso I1

Clase 12

IIC 2133 - Sección 1

Prof. Diego Arroyuelo

Sumario

Interrogación 1

Objetivos a evaluar en la I1

Interrogación 1

Objetivos a evaluar en la I1

- ☐ Comprender y comparar algoritmos de ordenación clásicos

Interrogación 1

Objetivos a evaluar en la I1

- ☐ Comprender y comparar algoritmos de ordenación clásicos
- ☐ Modificar algoritmos conocidos para resolver problemas

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

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

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

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

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

Interrogación 1

Formato de la prueba

- 2 horas de tiempo

Interrogación 1

Formato de la prueba

- 2 horas de tiempo
- Pool de 4 preguntas para elegir 3

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

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!

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

Interrogación 1

Material adicional

- Pueden usar un formulario/apuntes durante la prueba

Interrogación 1

Material adicional

- Pueden usar un formulario/apuntes durante la prueba
- Debe estar escrito a mano

Interrogación 1

Material adicional

- Pueden usar un formulario/apuntes durante la prueba
- Debe estar escrito a mano
- Una hoja carta (por ambos lados)

Interrogación 1

Material adicional

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

Interrogación 1

Material adicional

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

No se aceptarán diapositivas impresas

Sumario

SelectionSort

SelectionSort

Lema:

SelectionSort

Lema: *seleccionar de forma ordenada*

SelectionSort

Lema: *seleccionar de forma ordenada*

Funcionamiento a alto nivel

SelectionSort

Lema: *seleccionar de forma ordenada*

Funcionamiento a alto nivel

1. Seleccionar menor elemento

SelectionSort

Lema: *seleccionar de forma ordenada*

Funcionamiento a alto nivel

1. Seleccionar menor elemento
2. Ubicarlo como último elemento en zona *ordenada*

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

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

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

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

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 $A[0 \ldots n-1]$

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 $\mathcal{O}(n)$

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*:

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

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

InsertionSort

InsertionSort

Lema:

InsertionSort

Lema: *insertar de forma ordenada*

InsertionSort

Lema: *insertar de forma ordenada*

Funcionamiento a alto nivel

InsertionSort

Lema: *insertar de forma ordenada*

Funcionamiento a alto nivel

1. Seleccionar el primer elemento no revisado

InsertionSort

Lema: *insertar de forma ordenada*

Funcionamiento a alto nivel

1. Seleccionar el primer elemento no revisado
2. Moverlo hasta su posición correcta

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

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

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...

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

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 $A[0 \ldots n-1]$

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 $A[0 \ldots n-1]$

- Mejor caso: secuencia ordenada

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 $\mathcal{O}(1)$

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:

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 $i = 1 \dots n-1$ **:**
        $j = i$
        while $(j > 0) \land (A[j] < A[j-1])$ **:**
            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

MergeSort

Lema:

MergeSort

Lema: *mezclar mitades ordenadas*

MergeSort

Lema: *mezclar mitades ordenadas*

Funcionamiento a alto nivel

MergeSort

Lema: *mezclar mitades ordenadas*

Funcionamiento a alto nivel

1. Recursivamente dividir secuencia en mitades

MergeSort

Lema: *mezclar mitades ordenadas*

Funcionamiento a alto nivel

1. Recursivamente dividir secuencia en mitades
2. Caso base: secuencias de largo 1

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`)

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

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

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

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 $A[0 \ldots n-1]$

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 ← MergeSort(A_1)
    B_2 ← MergeSort(A_2)
    B ← Merge(B_1, B_2)
    return B
```

```
Merge(A, B):
    Iniciar C vacía
    while |A| > 0 ∧ |B| > 0 :
        if A[1] ≤ B[1] :
            e ← Extraer(A[1])
        else:
            e ← 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

QuickSort

Lema:

QuickSort

Lema: *ordenar pivotes de forma recursiva*

QuickSort

Lema: *ordenar pivotes de forma recursiva*

Funcionamiento a alto nivel

QuickSort

Lema: *ordenar pivotes de forma recursiva*

Funcionamiento a alto nivel

1. Recursivamente ordenar un elemento arbitrario (pivote)

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

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

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

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

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

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 $A[0 \dots n-1]$

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

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 $\mathcal{O}(1)$

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

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 $\{i,\ldots,f\}$; $p \leftarrow A[x]$

  $A[x] \rightleftarrows A[f]$

  $j \leftarrow i$

  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`

Mejoras en `Quicksort`

- Para sub-secuencias pequeñas (e.g. $n \leq 20$) podemos usar `InsertionSort`

Mejoras en `Quicksort`

- Para sub-secuencias pequeñas (e.g. $n \leq 20$) podemos usar `InsertionSort`
  - A pesar de no tener una complejidad mejor

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

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

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

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

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

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 $A[k_1], A[k_2], A[k_3]$

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

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

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

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

Heaps binarios

Definición

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

Heaps binarios

Definición

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

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

Heaps binarios

Definición

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

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

Heaps binarios

Definición

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

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

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**

Heaps binarios

Definición

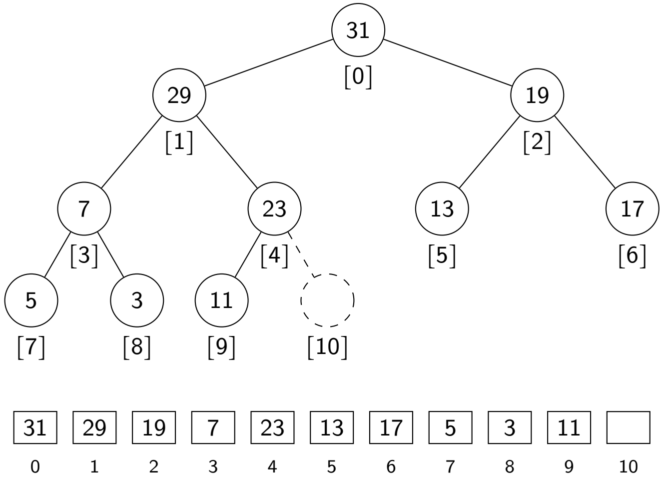
Un **heap binario** H es un árbol binario tal que

- $H.\mathit{left}$ y $H.\mathit{right}$ son heaps binarios
- $H.\mathit{value} > H.\mathit{left}.\mathit{value}$
- $H.\mathit{value} > H.\mathit{right}.\mathit{value}$

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

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

Heaps binarios: representación compacta



Heaps binarios

Aspectos esenciales

Heaps binarios

Aspectos esenciales

- Propiedad de heap

Heaps binarios

Aspectos esenciales

- Propiedad de heap

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

Heaps binarios

Aspectos esenciales

- Propiedad de heap

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

- Se asegura **balance** mediante sus operaciones

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`)

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`)

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`)

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

Heaps binarios

Operaciones

- Inserción: agregar un nuevo elemento

Heaps binarios

Operaciones

- Inserción: agregar un nuevo elemento
  - Se inserta al final del arreglo

Heaps binarios

Operaciones

- Inserción: agregar un nuevo elemento
  - Se inserta al final del arreglo
  - Se reubica con `SiftUp`

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

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

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`

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

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`

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

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

Heaps binarios

Orientaciones para el estudio

- Comprender la representación compacta de heaps

Heaps binarios

Orientaciones para el estudio

- Comprender la representación compacta de heaps
- Comprender operaciones y su uso para mantener una cola de prioridad

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?

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

Orden lineal

Aspectos esenciales de los algoritmos de orden lineal

- Operan sobre naturales (índices para un arreglo)

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

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

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

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

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

Orden lineal

Orientaciones para el estudio

- Comprender cuándo se pueden ocupar los algoritmos lineales

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**

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

Dividir para conquistar

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo

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

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

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 ← ⌊(i + f) / 2⌋
    if A[m] = x : return m
    if A[m] > x :
        return BSearch (A, x, i, m − 1)
    return BSearch (A, x, m + 1, f)
```

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 ← ⌊(i + f)/2⌋
    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

Diccionarios

Definición

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

- **Asociar** un valor a una llave

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

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

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

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

Diccionarios: dos enfoques

Vimos dos instancias de diccionarios

1. Árboles de búsqueda

Diccionarios: dos enfoques

Vimos dos instancias de diccionarios

1. Árboles de búsqueda
   - Árbol Binario
   - Binarios AVL

Diccionarios: dos enfoques

Vimos dos instancias de diccionarios

1. Árboles de búsqueda
    - Árbol Binario
    - Binarios AVL

Diccionarios: dos enfoques

Vimos dos instancias de diccionarios

1. Árboles de búsqueda
   - Árbol Binario
   - Binarios AVL

Árboles de búsqueda

Tipos estudiados: binarios, AVL

Aspectos esenciales

Árboles de búsqueda

Tipos estudiados: binarios, AVL

Aspectos esenciales

- Todos los tipos tienen la **propiedad de búsqueda**: valores ordenados en

Árboles de búsqueda

Tipos estudiados: binarios, AVL

Aspectos esenciales

- Todos los tipos tienen la **propiedad de búsqueda**: valores ordenados en

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

Árboles de búsqueda

Tipos estudiados: binarios, AVL

Aspectos esenciales

- Todos los tipos tienen la **propiedad de búsqueda**: valores ordenados en

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

- AVL tipo tiene una noción de **balance** dada por sus operaciones

Árboles de búsqueda

Tipos estudiados: binarios, AVL

Aspectos esenciales

- Todos los tipos tienen la **propiedad de búsqueda**: valores ordenados en

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

- AVL tipo tiene una noción de **balance** dada por sus operaciones
  - AVL cuida la altura de sus hijos recursivamente

Árboles de búsqueda

Tipos estudiados: binarios, AVL

Aspectos esenciales

- Todos los tipos tienen la **propiedad de búsqueda**: valores ordenados en

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

- AVL tipo tiene una noción de **balance** dada por sus operaciones
    - AVL cuida la altura de sus hijos recursivamente

Árboles de búsqueda

Tipos estudiados: binarios, AVL

Aspectos esenciales

- Todos los tipos tienen la **propiedad de búsqueda**: valores ordenados en

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

- AVL tipo tiene una noción de **balance** dada por sus operaciones
  - AVL cuida la altura de sus hijos recursivamente

Importancia del balance: mantener el árbol con profundidad logarítmica

Árboles de búsqueda

Operaciones

Árboles de búsqueda

Operaciones

- Búsqueda gracias a la propiedad de búsqueda

Árboles de búsqueda

Operaciones

- Búsqueda gracias a la propiedad de búsqueda
- Inserción

Árboles de búsqueda

Operaciones

- Búsqueda gracias a la propiedad de búsqueda
- Inserción
  - AVL: involucra posibles rotaciones

Árboles de búsqueda

Operaciones

- Búsqueda gracias a la propiedad de búsqueda
- Inserción
  - AVL: involucra posibles rotaciones
  - 2-3: involucra posibles splits

Árboles de búsqueda

Operaciones

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

Árboles de búsqueda

Operaciones

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

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

Árboles de búsqueda

Operaciones

- Búsqueda gracias a la propiedad de búsqueda
- Inserción
  - AVL: involucra posibles rotaciones
  - 2-3: involucra posibles splits
- 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

Árboles de búsqueda

Orientaciones para el estudio

- ☐ Comprender la estrategia de balance
- ☐ Construir árboles pequeños con inserción de llaves sucesivas

Árboles de búsqueda

Orientaciones para el estudio

- ☐ Comprender la estrategia de balance
- ☐ Construir árboles pequeños con inserción de llaves sucesivas
- ☐ Definir pequeñas secuencias de inserción que generan árboles más/menos desbalanceados

Árboles de búsqueda

Orientaciones para el estudio

- ☐ Comprender la estrategia de balance
- ☐ 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?

Árboles de búsqueda

Orientaciones para el estudio

- ☐ Comprender la estrategia de balance
- ☐ 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

5

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 $A[0, \dots, n-1]$ e índices i, f

output: Lista de pares de índices L

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

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

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 $A[0 \dots n-1]$ 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):
begin
    if (f − i) = 0 :
    begin
        if A[i] = i :
        begin
            return i
        end
        return null
    end
    p ← ⌊(f − i)/2⌋
    if A[p] = p :
    begin
        return p
    end
    if A[p] > p :
    begin
```

Sumario

Recomendaciones finales

Recomendaciones finales

Para los algoritmos vistos en clase

Recomendaciones finales

Para los algoritmos vistos en clase

- Comprender las demostraciones de correctitud

Recomendaciones finales

Para los algoritmos vistos en clase

- Comprender las demostraciones de correctitud
- Replicarlas por su cuenta

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?*

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

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

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!

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

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