Complejidad Algorítmica

Introducción

Ricardo Eugenio González Valenzuela

Segundo Semestre, 2020

Agradecimentos

El material de este curso está basado en el material del curso de *Análisis de Algorítmos* ministrado por la *Universidad Estatal de Campinas*. Un agradecimiento especial al Prof. Dr. Lehilton Lelis Chaves Pedrosa.

El conjunto de diapositivas de cada unidad será disponibilizada como guía de estudios y debe ser usada únicamente para revisar las aulas. Para estudiar, y practicar, *lea* el libro-texto *Introducción a los Algoritmos - Thomas Cormen et al.*, y resuelva los ejercícios sugeridos.

Ricardo.

Introducción a la Complejidad Algorítmica

Objetivos

¿Qué veremos en esta disciplina?

1. Demostrar correctitud de un algoritmo
   - como tener certeza de que la salida es correcta
   - como convencer a otras personas de esto
2. Analizar la complejidad de un algoritmo
   - como estimar la cantidad de recursos utilizados
   - los recursos pueden ser tiempo, memoria, acceso a la red, etc.

Objetivos

¿Qué veremos en esta disciplina?

1. Demostrar correctitud de un algoritmo
   - como tener certeza de que la salida es correcta
   - como convencer a otras personas de esto

2. Analizar la complejidad de un algoritmo
   - como estimar la cantidad de recursos utilizados
   - los recursos pueden ser tiempo, memoria, acceso a la red, etc.

Objetivos

¿Qué veremos en esta disciplina?

1. Demostrar correctitud de un algoritmo
   - como tener certeza de que la salida es correcta
   - como convencer a otras personas de esto

2. Analizar la complejidad de un algoritmo
   - como estimar la cantidad de recursos utilizados
   - los recursos pueden ser tiempo, memoria, acceso a la red, etc.

Objetivos

¿Qué veremos en esta disciplina?

1. Demostrar correctitud de un algoritmo
   - como tener certeza de que la salida es correcta
   - como convencer a otras personas de esto

2. Analizar la complejidad de un algoritmo
   - como estimar la cantidad de recursos utilizados
   - los recursos pueden ser tiempo, memoria, acceso a la red, etc.

Objetivos

¿Qué veremos en esta disciplina?

1. Demostrar correctitud de un algoritmo
   - como tener certeza de que la salida es correcta
   - como convencer a otras personas de esto

2. Analizar la complejidad de un algoritmo
   - como estimar la cantidad de recursos utilizados
   - los recursos pueden ser tiempo, memoria, acceso a la red, etc.

Objetivos

¿Qué veremos en esta disciplina?

1. Demostrar correctitud de un algoritmo
   - como tener certeza de que la salida es correcta
   - como convencer a otras personas de esto

2. Analizar la complejidad de un algoritmo
   - como estimar la cantidad de recursos utilizados
   - los recursos pueden ser tiempo, memoria, acceso a la red, etc.

Objetivos

¿Qué veremos en esta disciplina?

1. Demostrar correctitud de un algoritmo
   - como tener certeza de que la salida es correcta
   - como convencer a otras personas de esto

2. Analizar la complejidad de un algoritmo
   - como estimar la cantidad de recursos utilizados
   - los recursos pueden ser tiempo, memoria, acceso a la red, etc.

Objetivos

3. Utilizar técnicas conocidas de proyectos de algoritmos
   - divide y vencerás, programación dinámica, etc.
   - utilizar recursividad adecuadamente

4. Entender la dificultad intrínseca de algunos problemas
   - inexistencia de algoritmos eficientes
   - identificar los problemas intratables

Objetivos

3. Utilizar técnicas conocidas de proyectos de algoritmos
   - divide y vencerás, programación dinámica, etc.
   - utilizar recursividad adecuadamente

4. Entender la dificultad intrínseca de algunos problemas
   - inexistencia de algoritmos eficientes
   - identificar los problemas intratables

Objetivos

3. Utilizar técnicas conocidas de proyectos de algoritmos
   - divide y vencerás, programación dinámica, etc.
   - utilizar recursividad adecuadamente

4. Entender la dificultad intrínseca de algunos problemas
   - inexistencia de algoritmos eficientes
   - identificar los problemas intratables

Objetivos

3. Utilizar técnicas conocidas de proyectos de algoritmos
   - divide y vencerás, programación dinámica, etc.
   - utilizar recursividad adecuadamente

4. Entender la dificultad intrínseca de algunos problemas
   - inexistencia de algoritmos eficientes
   - identificar los problemas intratables

Objetivos

3. Utilizar técnicas conocidas de proyectos de algoritmos
   - divide y vencerás, programación dinámica, etc.
   - utilizar recursividad adecuadamente

4. Entender la dificultad intrínseca de algunos problemas
   - inexistencia de algoritmos eficientes
   - identificar los problemas intratables

Objetivos

3. Utilizar técnicas conocidas de proyectos de algoritmos
   - divide y vencerás, programación dinámica, etc.
   - utilizar recursividad adecuadamente

4. Entender la dificultad intrínseca de algunos problemas
   - inexistencia de algoritmos eficientes
   - identificar los problemas intratables

Problemas computacionales

Un problema computacional es una relación entre un conjunto de instancias y un conjunto de soluciones:

- una instancia es un conjunto de valores conocidos
- una solución es un conjunto de valores a calcular
- cada instancia corresponde a una o mas soluciones

Problemas computacionales

Un problema computacional es una relación entre un conjunto de instancias y un conjunto de soluciones:

- una instancia es un conjunto de valores conocidos
- una solución es un conjunto de valores a calcular
- cada instancia corresponde a una o mas soluciones

Problemas computacionales

Un problema computacional es una relación entre un conjunto de instancias y un conjunto de soluciones:

- una instancia es un conjunto de valores conocidos
- una solución es un conjunto de valores a calcular
- cada instancia corresponde a una o mas soluciones

Problemas computacionales

Un problema computacional es una relación entre un conjunto de instancias y un conjunto de soluciones:

- una instancia es un conjunto de valores conocidos
- una solución es un conjunto de valores a calcular
- cada instancia corresponde a una o mas soluciones

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: prueba de primalidad

Problema: determinar si un dado número es primo

- instancias: números enteros
- soluciones: sí o no

Ejemplo:

- Instancia: 9411461
- Solución: sí

Ejemplo:

- Instancia: 8411461
- Solución: no

Ejemplo de problema: ordenamiento

Problema: ordenar los elementos de un vector

- instancias: conjunto de vectores de enteros
- soluciones: conjunto de vectores de enteros en orden creciente

Ejemplo:

- Instancia:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solución:

1										n
11	22	22	33	33	33	44	55	55	77	99

Ejemplo de problema: ordenamiento

Problema: ordenar los elementos de un vector

- instancias: conjunto de vectores de enteros
- soluciones: conjunto de vectores de enteros en orden creciente

Ejemplo:

- Instancia:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solución:

1										n
11	22	22	33	33	33	44	55	55	77	99

Ejemplo de problema: ordenamiento

Problema: ordenar los elementos de un vector

- instancias: conjunto de vectores de enteros
- soluciones: conjunto de vectores de enteros en orden creciente

Ejemplo:

- Instancia:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solución:

1										n
11	22	22	33	33	33	44	55	55	77	99

Ejemplo de problema: ordenamiento

Problema: ordenar los elementos de un vector

- instancias: conjunto de vectores de enteros
- soluciones: conjunto de vectores de enteros en orden creciente

Ejemplo:

- Instancia:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solución:

1										n
11	22	22	33	33	33	44	55	55	77	99

Ejemplo de problema: ordenamiento

Problema: ordenar los elementos de un vector

- instancias: conjunto de vectores de enteros
- soluciones: conjunto de vectores de enteros en orden creciente

Ejemplo:

- Instancia:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solución:

1										n
11	22	22	33	33	33	44	55	55	77	99

Ejemplo de problema: ordenamiento

Problema: ordenar los elementos de un vector

- instancias: conjunto de vectores de enteros
- soluciones: conjunto de vectores de enteros en orden creciente

Ejemplo:

- Instancia:

1										n
33	55	33	44	33	22	11	99	22	55	77

- Solución:

1										n
11	22	22	33	33	33	44	55	55	77	99

Algoritmos

Un algoritmo es una secuencia de instrucciones que

- recibe una instancia de un problema computacional
- devuelve una solución correspondiente a la instancia recibida

Observaciones:

- la instancia recibida es llamada entrada
- la solución devuelta es llamada de salida
- toda instrucción debe ser bien definida

Algoritmos

Un algoritmo es una secuencia de instrucciones que

- recibe una instancia de un problema computacional
- devuelve una solución correspondiente a la instancia recibida

Observaciones:

- la instancia recibida es llamada entrada
- la solución devuelta es llamada de salida
- toda instrucción debe ser bien definida

Algoritmos

Un algoritmo es una secuencia de instrucciones que

- recibe una instancia de un problema computacional
- devuelve una solución correspondiente a la instancia recibida

Observaciones:

- la instancia recibida es llamada entrada
- la solución devuelta es llamada de salida
- toda instrucción debe ser bien definida

Algoritmos

Un algoritmo es una secuencia de instrucciones que

- recibe una instancia de un problema computacional
- devuelve una solución correspondiente a la instancia recibida

Observaciones:

- la instancia recibida es llamada entrada
- la solución devuelta es llamada de salida
- toda instrucción debe ser bien definida

Algoritmos

Un algoritmo es una secuencia de instrucciones que

- recibe una instancia de un problema computacional
- devuelve una solución correspondiente a la instancia recibida

Observaciones:

- la instancia recibida es llamada entrada
- la solución devuelta es llamada de salida
- toda instrucción debe ser bien definida

Algoritmos

Un algoritmo es una secuencia de instrucciones que

- recibe una instancia de un problema computacional
- devuelve una solución correspondiente a la instancia recibida

Observaciones:

- la instancia recibida es llamada entrada
- la solución devuelta es llamada de salida
- toda instrucción debe ser bien definida

Algoritmos

Un algoritmo es una secuencia de instrucciones que

- recibe una instancia de un problema computacional
- devuelve una solución correspondiente a la instancia recibida

Observaciones:

- la instancia recibida es llamada entrada
- la solución devuelta es llamada de salida
- toda instrucción debe ser bien definida

Descripción de algoritmos

Podemos escribir un algoritmo de varias maneras:

- en un lenguaje de programación, como C, Pascal, Java, Python...
- en español, o en otra lengua natural
- en pseudocódigo, como en el libro de Introducción a los Algoritmos

Durante las clases iremos intercalando entre estas tres opciones

Descripción de algoritmos

Podemos escribir un algoritmo de varias maneras:

- en un lenguaje de programación, como C, Pascal, Java, Python...
- en español, o en otra lengua natural
- en pseudocódigo, como en el libro de Introducción a los Algoritmos

Durante las clases iremos intercalando entre estas tres opciones

Descripción de algoritmos

Podemos escribir un algoritmo de varias maneras:

- en un lenguaje de programación, como C, Pascal, Java, Python...
- en español, o en otra lengua natural
- en pseudocódigo, como en el libro de Introducción a los Algoritmos

Durante las clases iremos intercalando entre estas tres opciones

Descripción de algoritmos

Podemos escribir un algoritmo de varias maneras:

- en un lenguaje de programación, como C, Pascal, Java, Python...
- en español, o en otra lengua natural
- en pseudocódigo, como en el libro de Introducción a los Algoritmos

Durante las clases iremos intercalando entre estas tres opciones

Descripción de algoritmos

Podemos escribir un algoritmo de varias maneras:

- en un lenguaje de programación, como C, Pascal, Java, Python...
- en español, o en otra lengua natural
- en pseudocódigo, como en el libro de Introducción a los Algoritmos

Durante las clases iremos intercalando entre estas tres opciones

Ejemplo de pseudocódigo

Un algoritmo para el problema de ordenamiento:

```
INSERTION-SORT(A, n)
1   para j ← 2 hasta n hacer
2       llave ← A[j]
3       i ← j − 1
4       mientras i ≥ 1 y A[i] > llave hacer
5           A[i + 1] ← A[i]
6           i ← i − 1
7       A[i + 1] ← llave
```

Malas prácticas

- No mezcle código con pseudocódigo:

```
for (i = 0; i < n; i++) ...
```

```
if (A[i] >= llave)
    break
```

- No escriba frases confusas:

```
si A[i] > llave entonces
    intercambie las posiciones de los elementos
```

```
llave ← obtiene el próximo elemento
i ← busca la posición de la llave
```

- Evite algoritmos complicados o con muchas variables:

```
si j = 1 entonces
    A[2] ← A[1]
    A[1] ← llave
sino
    mientras i ≥ 1 e A[i] ≥ llave hacer
        ...
```

Modelo computacional

Sólo podemos escribir instrucciones bien definidas:

- el resultado de cada instrucción no es ambíguo y depende solamente del estado actual de la ejecución
- debe ser posible ejecutar cada instrucción usando la computadora adoptada

El conjunto de instrucciones permitidas está determinado por lo que llamamos modelo computacional

Modelo computacional

Sólo podemos escribir instrucciones bien definidas:

- el resultado de cada instrucción no es ambíguo y depende solamente del estado actual de la ejecución
- debe ser posible ejecutar cada instrucción usando la computadora adoptada

El conjunto de instrucciones permitidas está determinado por lo que llamamos modelo computacional

Modelo computacional

Sólo podemos escribir instrucciones bien definidas:

- el resultado de cada instrucción no es ambíguo y depende solamente del estado actual de la ejecución
- debe ser posible ejecutar cada instrucción usando la computadora adoptada

El conjunto de instrucciones permitidas está determinado por lo que llamamos modelo computacional

Modelo computacional

Sólo podemos escribir instrucciones bien definidas:

- el resultado de cada instrucción no es ambíguo y depende solamente del estado actual de la ejecución
- debe ser posible ejecutar cada instrucción usando la computadora adoptada

El conjunto de instrucciones permitidas está determinado por lo que llamamos modelo computacional

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Correctitud de algoritmos

Um algoritmo es correcto si:

1. sólo utiliza instrucciones del modelo de computación adoptado,
2. termina para toda instancia del problema y
3. devuelve una solución que correspondente a la instancia recibida.

Al escribir un algoritmo, siempre debemos

1. Probar el algoritmo
   - con una o mas instancias de ejemplo
   - ejecutando o simulando el algoritmo
2. demostrar que el algoritmo está correcto
   - escribir una prueba formal genérica
   - vale para toda instancia del problema

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Complejidad Algorítmica

- No siempre es viable ejecutar un algoritmo
  - puede tardar años o siglos para terminar
  - puede necesitar mas memoria de la disponible

- Queremos
  1. proyectar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para esto, necesitamos medir a complejidad del algoritmo
  - independentemente de quien lo programo,
  - del lenguaje en fue escrito
  - de la máquina que será usada

Modelo computacional y complejidad

Vamos a estimar el tiempo de ejecución de un algoritmo

- contamos el número de instrucciones elementares ejecutadas
- suponemos que cada instrucción elemental consume un tiempo constante

El análisis de complejidad depende siempre del modelo computacional adoptado!

- queremos un modelo computacional realista
- el conjunto de instrucciones elementales debe ser compatible con las computadoras modernas
- pero debe ser suficientemente genérico para las diferentes arquiteturas

Modelo computacional y complejidad

Vamos a estimar el tiempo de ejecución de un algoritmo

- contamos el número de instrucciones elementares ejecutadas
- suponemos que cada instrucción elemental consume un tiempo constante

El análisis de complejidad depende siempre del modelo computacional adoptado!

- queremos un modelo computacional realista
- el conjunto de instrucciones elementales debe ser compatible con las computadoras modernas
- pero debe ser suficientemente genérico para las diferentes arquiteturas

Modelo computacional y complejidad

Vamos a estimar el tiempo de ejecución de un algoritmo

- contamos el número de instrucciones elementares ejecutadas
- suponemos que cada instrucción elemental consume un tiempo constante

El análisis de complejidad depende siempre del modelo computacional adoptado!

- queremos un modelo computacional realista
- el conjunto de instrucciones elementales debe ser compatible con las computadoras modernas
- pero debe ser suficientemente genérico para las diferentes arquiteturas

Modelo computacional y complejidad

Vamos a estimar el tiempo de ejecución de un algoritmo

- contamos el número de instrucciones elementares ejecutadas
- suponemos que cada instrucción elemental consume un tiempo constante

El análisis de complejidad depende siempre del modelo computacional adoptado!

- queremos un modelo computacional realista
- el conjunto de instrucciones elementales debe ser compatible con las computadoras modernas
- pero debe ser suficientemente genérico para las diferentes arquiteturas

Modelo computacional y complejidad

Vamos a estimar el tiempo de ejecución de un algoritmo

- contamos el número de instrucciones elementares ejecutadas
- suponemos que cada instrucción elemental consume un tiempo constante

El análisis de complejidad depende siempre del modelo computacional adoptado!

- queremos un modelo computacional realista
- el conjunto de instrucciones elementales debe ser compatible con las computadoras modernas
- pero debe ser suficientemente genérico para las diferentes arquiteturas

Modelo computacional y complejidad

Vamos a estimar el tiempo de ejecución de un algoritmo

- contamos el número de instrucciones elementares ejecutadas
- suponemos que cada instrucción elemental consume un tiempo constante

El análisis de complejidad depende siempre del modelo computacional adoptado!

- queremos un modelo computacional realista
- el conjunto de instrucciones elementales debe ser compatible con las computadoras modernas
- pero debe ser suficientemente genérico para las diferentes arquiteturas

Modelo computacional y complejidad

Vamos a estimar el tiempo de ejecución de un algoritmo

- contamos el número de instrucciones elementares ejecutadas
- suponemos que cada instrucción elemental consume un tiempo constante

El análisis de complejidad depende siempre del modelo computacional adoptado!

- queremos un modelo computacional realista
- el conjunto de instrucciones elementales debe ser compatible con las computadoras modernas
- pero debe ser suficientemente genérico para las diferentes arquiteturas

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (`si`, `mientras`...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (`si`, `mientras`...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (si, mientras...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (si, mientras...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (si, mientras...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (si, mientras...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (si, mientras...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (si, mientras...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (**si**, **mientras**...)

Operaciones como la potencia no son elementales.

Máquinas RAM

Usaremos el Modelo Abstracto RAM (Random Access Machine)

- simula máquinas convencionales
- posee un único processador secuencial
- los tipos básicos son números enteros y puntos flotantes
- cada *palabra de memoria* tiene tamaño limitado, i.e.: los valores no pueden ser arbitrários

Instrucciones elementales

- operaciones aritméticas como suma, resta, producto...
- accesso directo a las posiciones de la memoria
- estructuras de control selectivas y repetitivas (**si**, **mientras**...)

Operaciones como la potencia no son elementales.

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Tamaño de la entrada

Un parámetro importante es el tamaño de la entrada:

- normalmente proporcional al número de bits de la entrada
- también usamos el número de elementos del vector

Problema: Primalidad

- Entrada: entero n
- Tamaño: $\lceil \log_2 n \rceil$ bits

Problema: Ordenamiento

- Entrada: vector $A[1 \ldots n]$
- Tamaño: $n\lceil \log_2 M \rceil$ bits, donde M es el máximo en $A[1 \ldots n]$

Análisis asintótico y del peor caso

Consideramos apenas instancias grandes

- el número de instrucciones normalmente crece con el tamaño de la entrada n
- instancias con tamaño limitado por constante gastan tiempos constantes

Hacemos apenas el análisis del peor caso

- restringimos la entrada con un dado tamaño n
- consideramos apenas una instancia para la cual el algoritmo ejecuta el mayor número de instrucciones

Denotamos por $T(n)$ el número de instrucciones ejecutadas en el peor caso para entradas de tamaño n

Análisis asintótico y del peor caso

Consideramos apenas instancias grandes

- el número de instrucciones normalmente crece con el tamaño de la entrada n
- instancias con tamaño limitado por constante gastan tiempos constantes

Hacemos apenas el análisis del peor caso

- restringimos la entrada con un dado tamaño n
- consideramos apenas una instancia para la cual el algoritmo ejecuta el mayor número de instrucciones

Denotamos por $T(n)$ el número de instrucciones ejecutadas en el peor caso para entradas de tamaño n

Análisis asintótico y del peor caso

Consideramos apenas instancias grandes

- el número de instrucciones normalmente crece con el tamaño de la entrada n
- instancias con tamaño limitado por constante gastan tiempos constantes

Hacemos apenas el análisis del peor caso

- restringimos la entrada con un dado tamaño n
- consideramos apenas una instancia para la cual el algoritmo ejecuta el mayor número de instrucciones

Denotamos por $T(n)$ el número de instrucciones ejecutadas en el peor caso para entradas de tamaño n

Análisis asintótico y del peor caso

Consideramos apenas instancias grandes

- el número de instrucciones normalmente crece con el tamaño de la entrada n
- instancias con tamaño limitado por constante gastan tiempos constantes

Hacemos apenas el análisis del peor caso

- restringimos la entrada con un dado tamaño n
- consideramos apenas una instancia para la cual el algoritmo ejecuta el mayor número de instrucciones

Denotamos por $T(n)$ el número de instrucciones ejecutadas en el peor caso para entradas de tamaño n

Análisis asintótico y del peor caso

Consideramos apenas instancias grandes

- el número de instrucciones normalmente crece con el tamaño de la entrada n
- instancias con tamaño limitado por constante gastan tiempos constantes

Hacemos apenas el análisis del peor caso

- restringimos la entrada con un dado tamaño n
- consideramos apenas una instancia para la cual el algoritmo ejecuta el mayor número de instrucciones

Denotamos por $T(n)$ el número de instrucciones ejecutadas en el peor caso para entradas de tamaño n

Análisis asintótico y del peor caso

Consideramos apenas instancias grandes

- el número de instrucciones normalmente crece con el tamaño de la entrada n
- instancias con tamaño limitado por constante gastan tiempos constantes

Hacemos apenas el análisis del peor caso

- restringimos la entrada con un dado tamaño n
- consideramos apenas una instancia para la cual el algoritmo ejecuta el mayor número de instrucciones

Denotamos por $T(n)$ el número de instrucciones ejecutadas en el peor caso para entradas de tamaño n

Análisis asintótico y del peor caso

Consideramos apenas instancias grandes

- el número de instrucciones normalmente crece con el tamaño de la entrada n
- instancias con tamaño limitado por constante gastan tiempos constantes

Hacemos apenas el análisis del peor caso

- restringimos la entrada con un dado tamaño n
- consideramos apenas una instancia para la cual el algoritmo ejecuta el mayor número de instrucciones

Denotamos por $T(n)$ el número de instrucciones ejecutadas en el peor caso para entradas de tamaño n

Características y limitaciones

Ese tipo de análisis de complejidad:

- normalmente estima bien el tiempo de ejecución real
- permite comparar diversos algoritmos para un problema
- continua siendo relevante aún con las evoluciones tecnológicas

Limitaciones:

- es una análisis pesimista del tiempo de execución
- en ciertas aplicaciones, ciertas instancias ocurren mas frecuentemente que el peor caso
- no provee información sobre tiempo de ejecución medio

Características y limitaciones

Ese tipo de análisis de complejidad:

- normalmente estima bien el tiempo de ejecución real
- permite comparar diversos algoritmos para un problema
- continua siendo relevante aún con las evoluciones tecnológicas

Limitaciones:

- es una análisis pesimista del tiempo de execución
- en ciertas aplicaciones, ciertas instancias ocurren mas frecuentemente que el peor caso
- no provee información sobre tiempo de ejecución medio

Características y limitaciones

Ese tipo de análisis de complejidad:

- normalmente estima bien el tiempo de ejecución real
- permite comparar diversos algoritmos para un problema
- continua siendo relevante aún con las evoluciones tecnológicas

Limitaciones:

- es una análisis pesimista del tiempo de execución
- en ciertas aplicaciones, ciertas instancias ocurren mas frecuentemente que el peor caso
- no provee información sobre tiempo de ejecución medio

Características y limitaciones

Ese tipo de análisis de complejidad:

- normalmente estima bien el tiempo de ejecución real
- permite comparar diversos algoritmos para un problema
- continua siendo relevante aún con las evoluciones tecnológicas

Limitaciones:

- es una análisis pesimista del tiempo de execución
- en ciertas aplicaciones, ciertas instancias ocurren mas frecuentemente que el peor caso
- no provee información sobre tiempo de ejecución medio

Características y limitaciones

Ese tipo de análisis de complejidad:

- normalmente estima bien el tiempo de ejecución real
- permite comparar diversos algoritmos para un problema
- continua siendo relevante aún con las evoluciones tecnológicas

Limitaciones:

- es una análisis pesimista del tiempo de execución
- en ciertas aplicaciones, ciertas instancias ocurren mas frecuentemente que el peor caso
- no provee información sobre tiempo de ejecución medio

Características y limitaciones

Ese tipo de análisis de complejidad:

- normalmente estima bien el tiempo de ejecución real
- permite comparar diversos algoritmos para un problema
- continua siendo relevante aún con las evoluciones tecnológicas

Limitaciones:

- es una análisis pesimista del tiempo de execución
- en ciertas aplicaciones, ciertas instancias ocurren mas frecuentemente que el peor caso
- no provee información sobre tiempo de ejecución medio

Características y limitaciones

Ese tipo de análisis de complejidad:

- normalmente estima bien el tiempo de ejecución real
- permite comparar diversos algoritmos para un problema
- continua siendo relevante aún con las evoluciones tecnológicas

Limitaciones:

- es una análisis pesimista del tiempo de execución
- en ciertas aplicaciones, ciertas instancias ocurren mas frecuentemente que el peor caso
- no provee información sobre tiempo de ejecución medio

Características y limitaciones

Ese tipo de análisis de complejidad:

- normalmente estima bien el tiempo de ejecución real
- permite comparar diversos algoritmos para un problema
- continua siendo relevante aún con las evoluciones tecnológicas

Limitaciones:

- es una análisis pesimista del tiempo de execución
- en ciertas aplicaciones, ciertas instancias ocurren mas frecuentemente que el peor caso
- no provee información sobre tiempo de ejecución medio

Comenzando a trabajar

Ordenamiento

Problema: ordenar los elementos de un vector

- Entrada: vector $A[1 \ldots n]$
- Salida: redistribución de $A[1 \ldots n]$ en orden creciente

Vamos a comenzar revisando el ordenamiento por inserción.

Ordenamiento

Problema: ordenar los elementos de un vector

- Entrada: vector $A[1 \ldots n]$
- Salida: redistribución de $A[1 \ldots n]$ en orden creciente

Vamos a comenzar revisando el ordenamiento por inserción.

Ordenamiento

Problema: ordenar los elementos de un vector

- Entrada: vector $A[1 \ldots n]$
- Salida: redistribución de $A[1 \ldots n]$ en orden creciente

Vamos a comenzar revisando el ordenamiento por inserción.

Ordenamiento

Problema: ordenar los elementos de un vector

- Entrada: vector $A[1 \ldots n]$
- Salida: redistribución de $A[1 \ldots n]$ en orden creciente

Vamos a comenzar revisando el ordenamiento por inserción.

Inserción en un vector ordenado

Idea del algoritmo

- suponga que el subvector $A[1 \ldots j-1]$ ya está ordenado.
- vamos a insertar $A[j]$ para que $A[1 \ldots j]$ quede ordenado
- el valor del elemento a ser insertado se llama *llave*

Antes de insertar:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Después de insertar:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserción en un vector ordenado

Idea del algoritmo

- suponga que el subvector $A[1 \ldots j-1]$ ya está ordenado.
- vamos a insertar $A[j]$ para que $A[1 \ldots j]$ quede ordenado
- el valor del elemento a ser insertado se llama *llave*

Antes de insertar:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Después de insertar:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserción en un vector ordenado

Idea del algoritmo

- suponga que el subvector $A[1 \ldots j-1]$ ya está ordenado.
- vamos a insertar $A[j]$ para que $A[1 \ldots j]$ quede ordenado
- el valor del elemento a ser insertado se llama *llave*

Antes de insertar:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Después de insertar:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserción en un vector ordenado

Idea del algoritmo

- suponga que el subvector $A[1 \ldots j-1]$ ya está ordenado.
- vamos a insertar $A[j]$ para que $A[1 \ldots j]$ quede ordenado
- el valor del elemento a ser insertado se llama *llave*

Antes de insertar:

Después de insertar:

Inserción en un vector ordenado

Idea del algoritmo

- suponga que el subvector $A[1 \ldots j-1]$ ya está ordenado.
- vamos a insertar $A[j]$ para que $A[1 \ldots j]$ quede ordenado
- el valor del elemento a ser insertado se llama *llave*

Antes de insertar:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Después de insertar:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Inserción en un vector ordenado

Idea del algoritmo

- suponga que el subvector $A[1 \ldots j-1]$ ya está ordenado.
- vamos a insertar $A[j]$ para que $A[1 \ldots j]$ quede ordenado
- el valor del elemento a ser insertado se llama *llave*

Antes de insertar:

1						j				n
20	25	35	40	44	55	38	99	10	65	50

Después de insertar:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

Insertando una llave

$$llave = 38$$

1					i	j				n
20	25	35	40	44	55	38	99	10	65	50

Insertando una llave

$llave = 38$

1					i	j				n
20	25	35	40	44	55	38	99	10	65	50

1				i		j				n
20	25	35	40	44		55	99	10	65	50

Insertando una llave

$llave = 38$

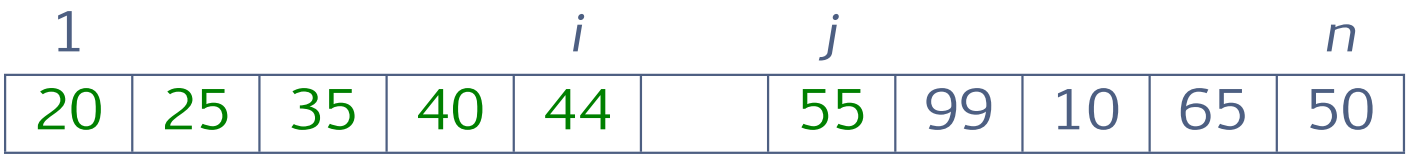
1					i	j				n
20	25	35	40	44	55	38	99	10	65	50

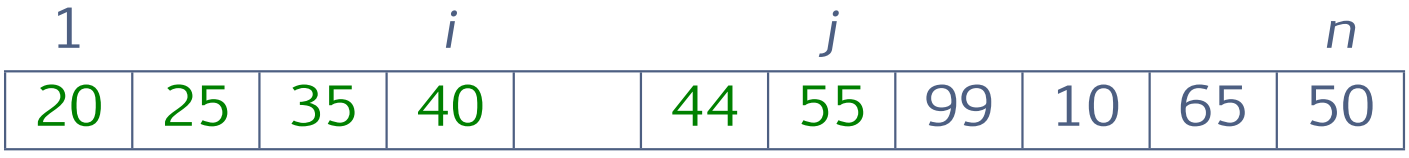
1				i		j				n
20	25	35	40	44		55	99	10	65	50

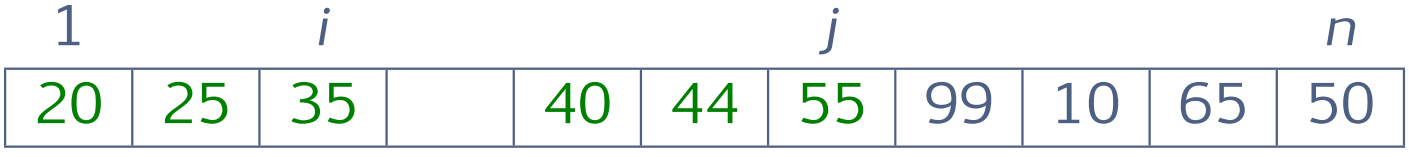
1			i			j				n
20	25	35	40		44	55	99	10	65	50

Insertando una llave

$llave = 38$

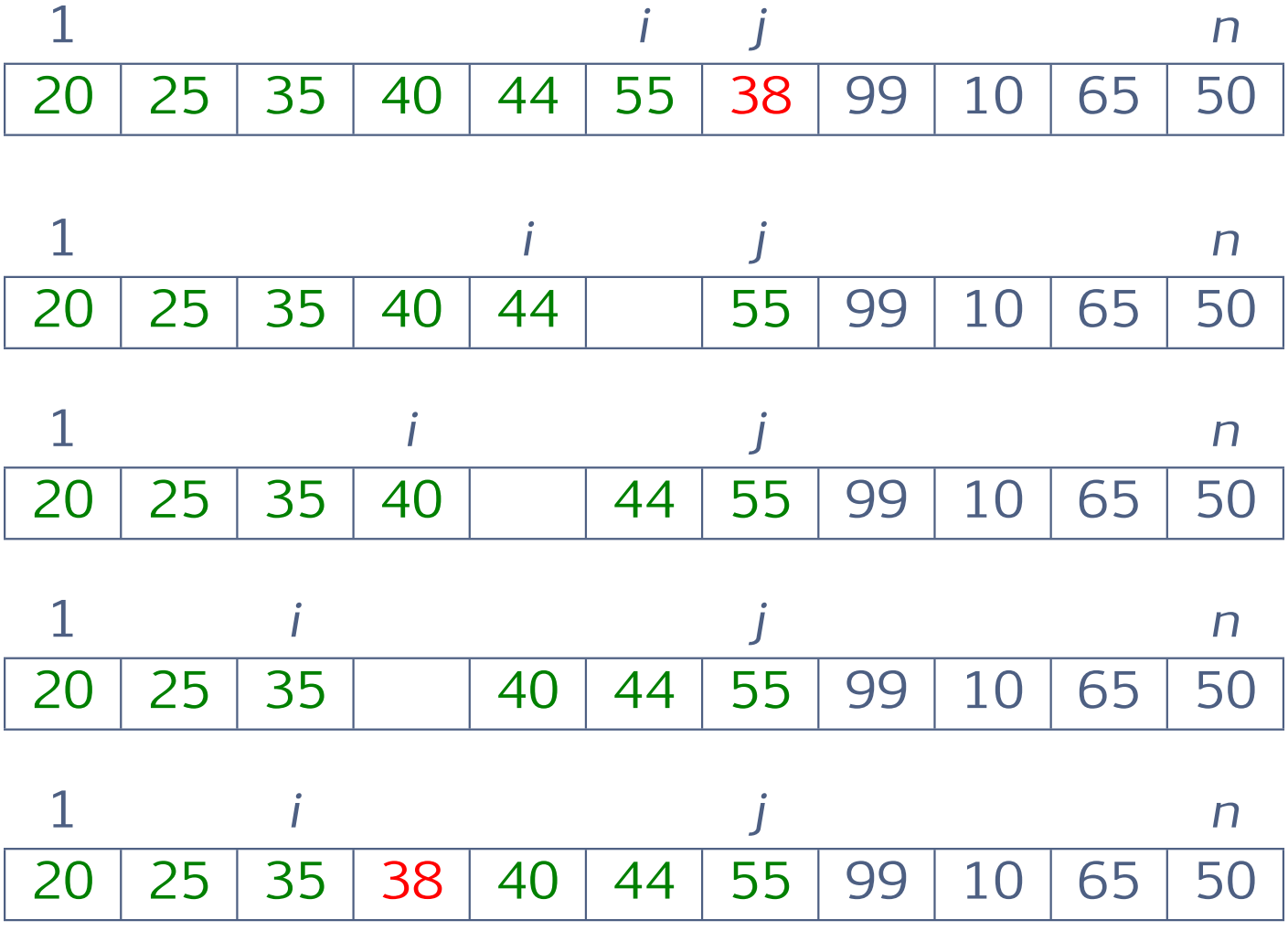




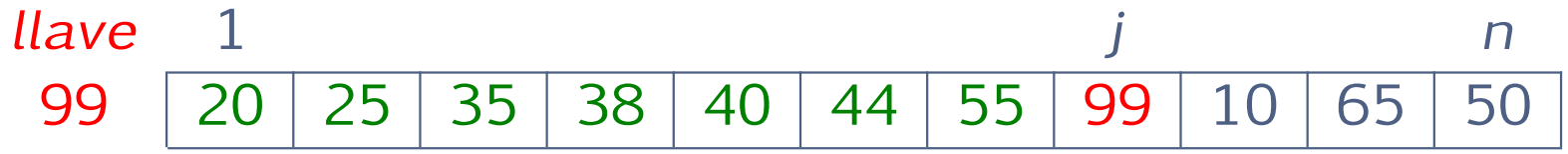


Insertando una llave

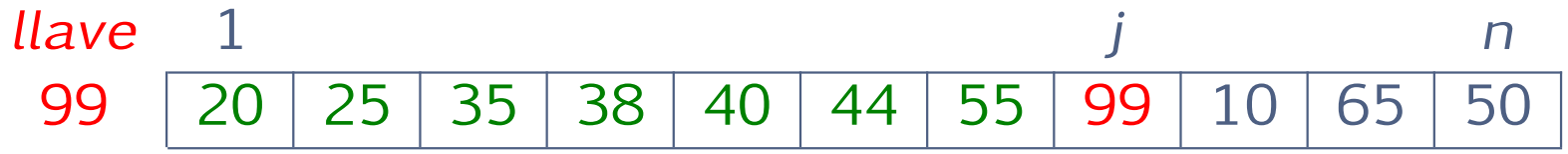
$llave = 38$



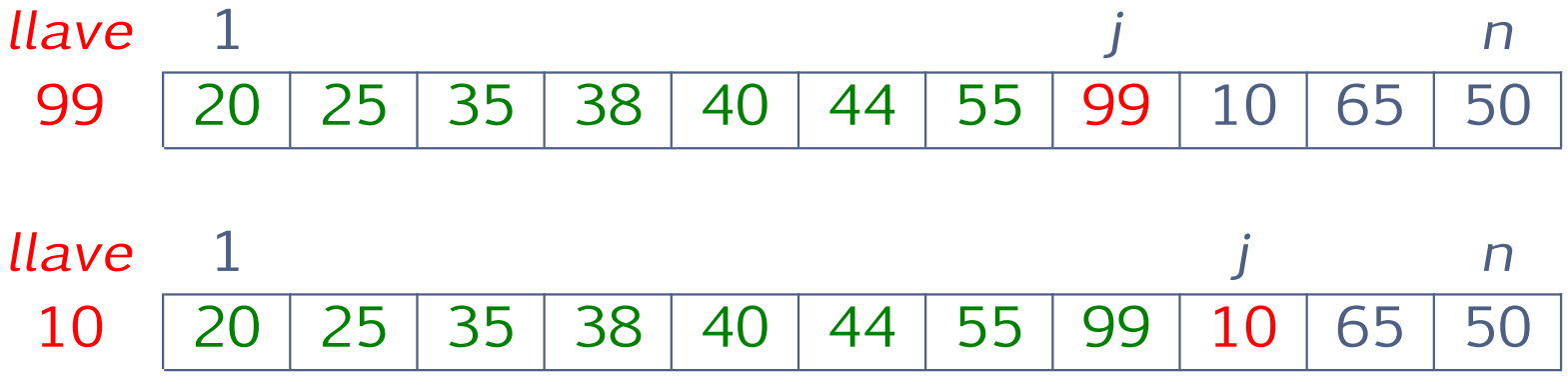
Ordenando por Inserción



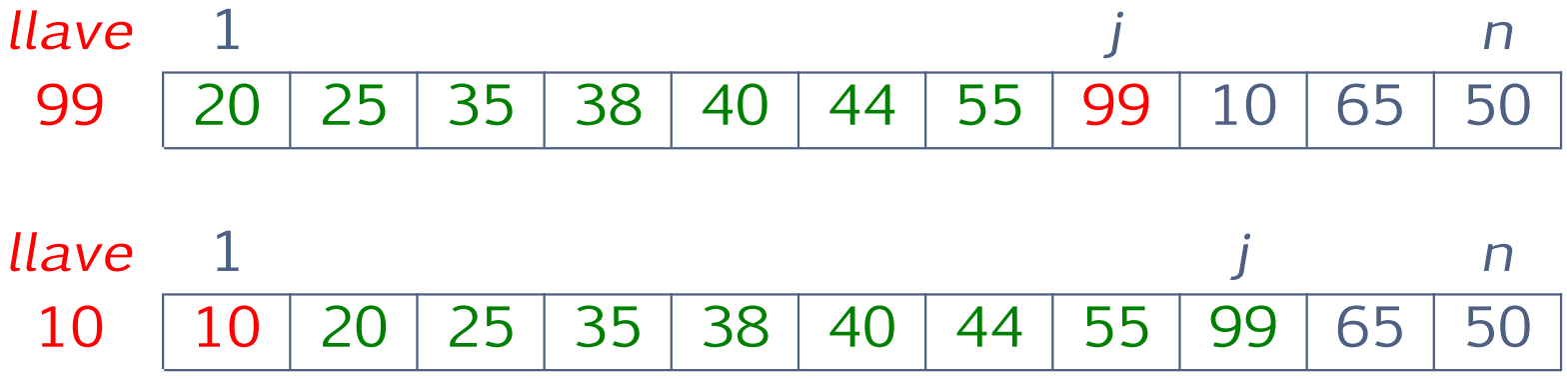
Ordenando por Inserción



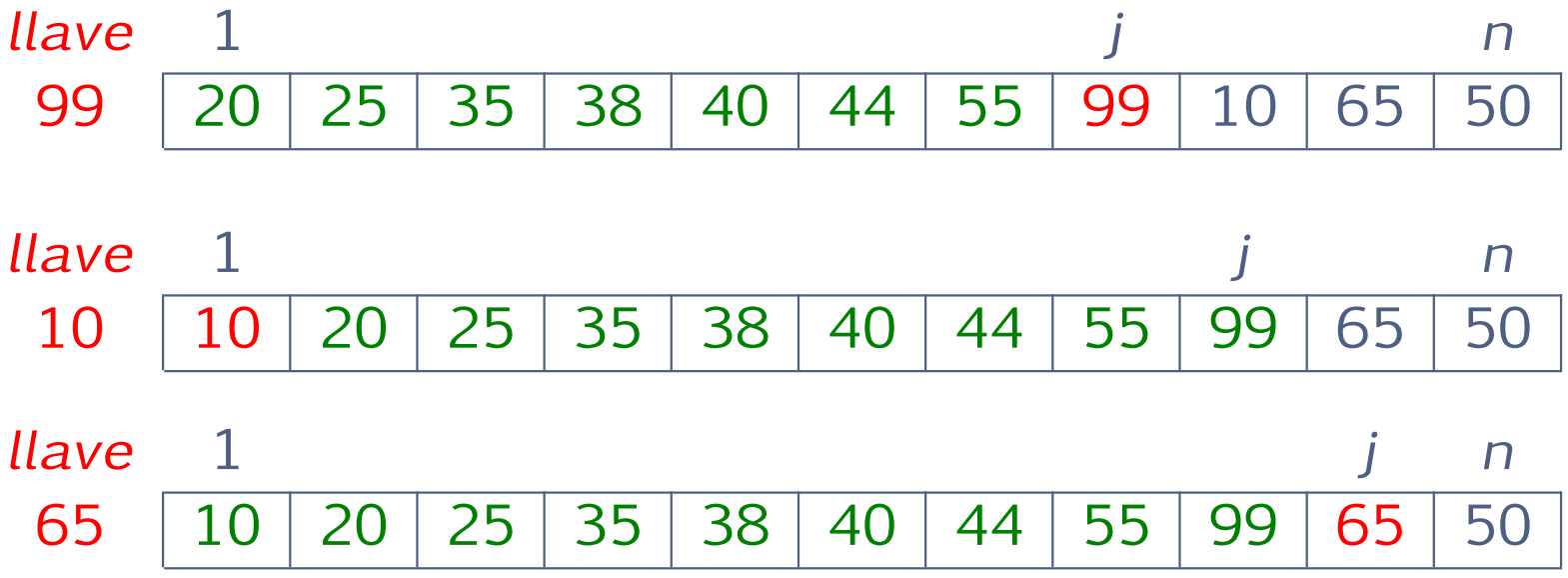
Ordenando por Inserción



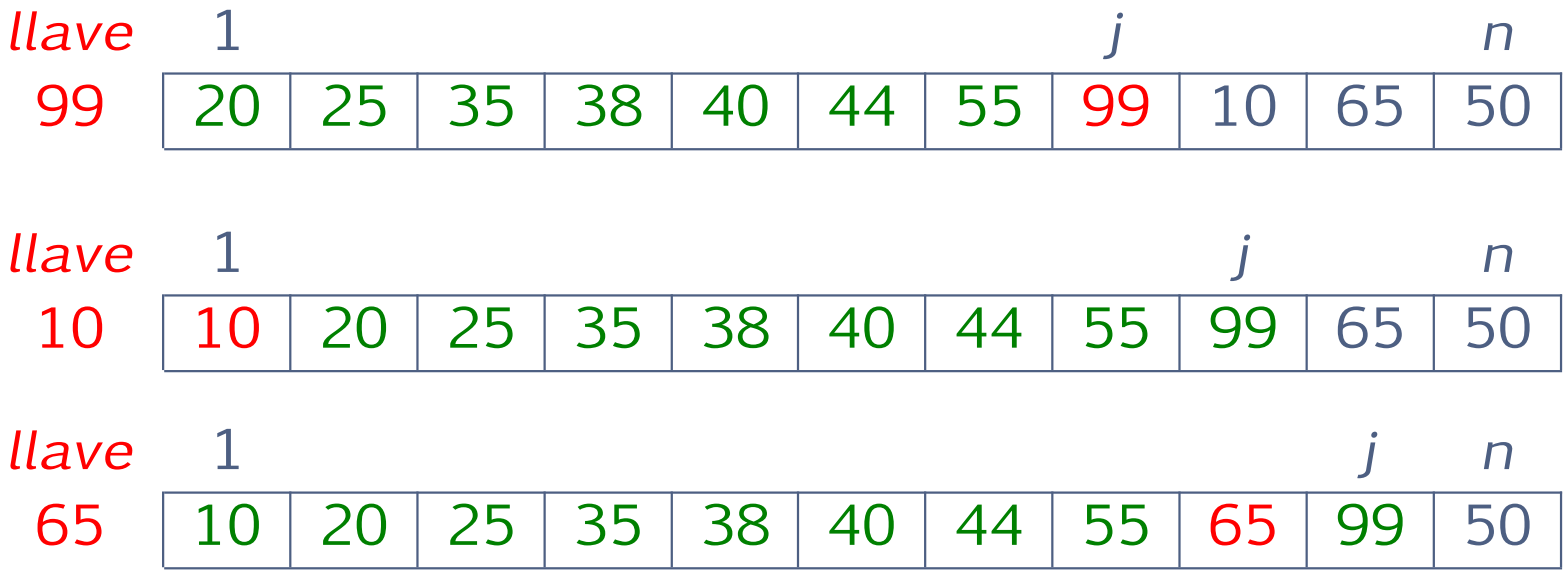
Ordenando por Inserción



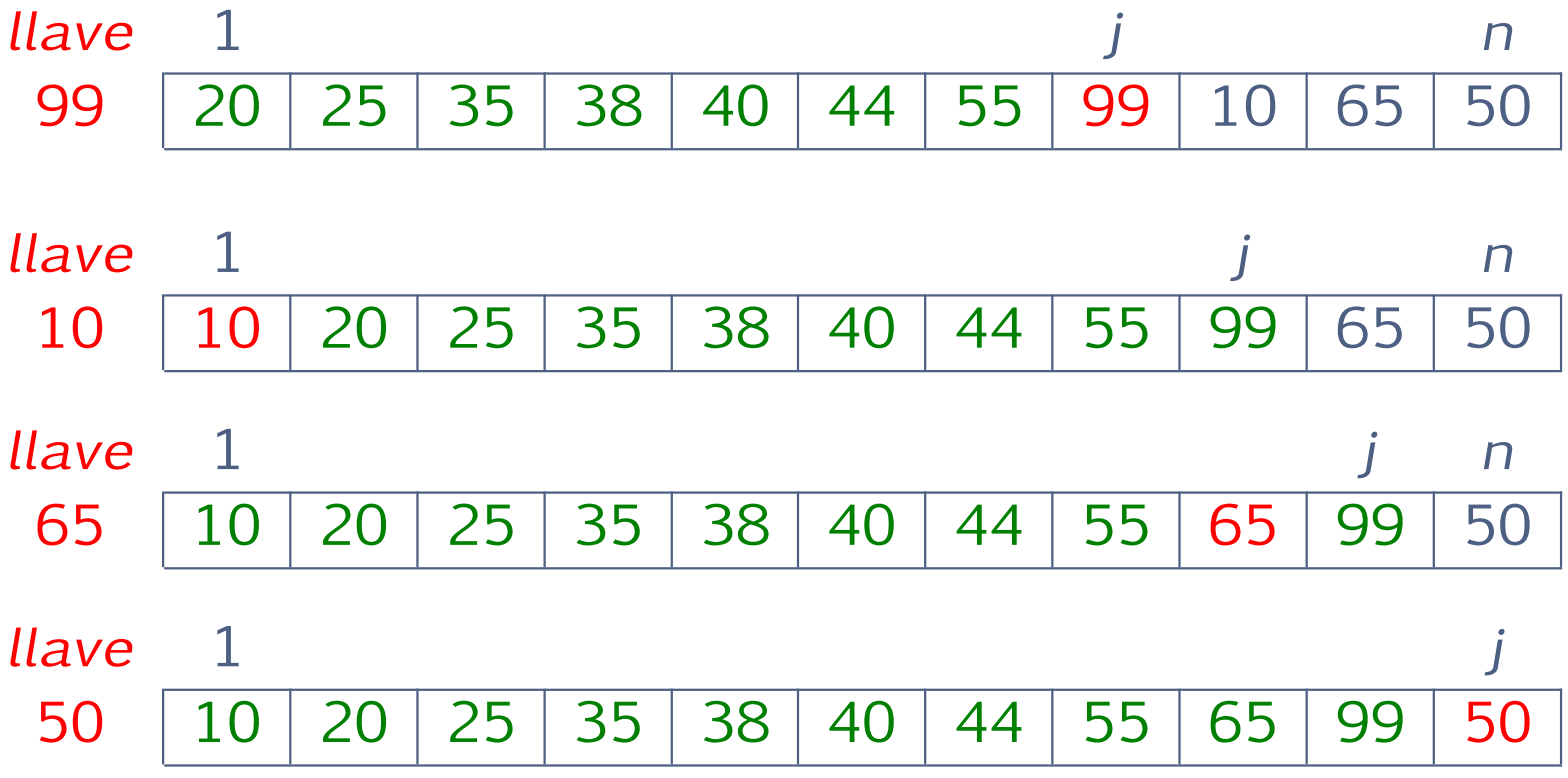
Ordenando por Inserción



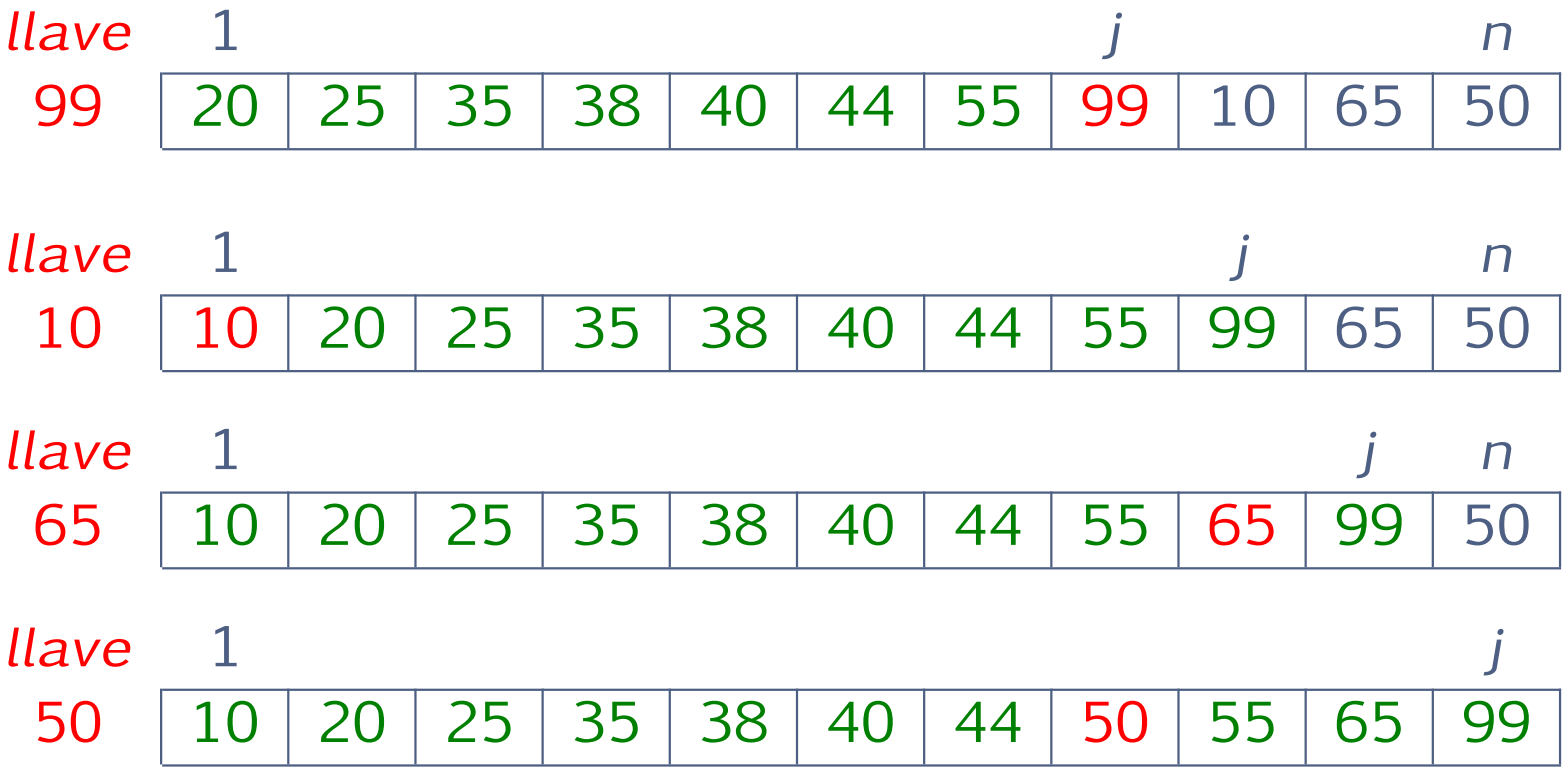
Ordenando por Inserción



Ordenando por Inserción



Ordenando por Inserción



Pseudocódigo de Insertion-Sort

```
INSERTION-SORT(A, n)
1   para j ← 2 hasta n hacer
2       llave ← A[j]
3       i ← j − 1
4       mientras i ≥ 1 y A[i] > llave hacer
5           A[i + 1] ← A[i]
6           i ← i − 1
7       A[i + 1] ← llave
```

Análisis del algoritmo

Qué es importante analizar?

- Correctitud
  - ya probamos el algoritmo con un ejemplo
  - mientras tanto, suponga que el algoritmo es correcto
  - después mostraremos que el algoritmo es correcto
- Complejidad de tiempo
  - considere vectores con n elementos
  - ¿cuántas instrucciones son ejecutadas?

Análisis del algoritmo

Qué es importante analizar?

- Correctitud
  - ya probamos el algoritmo con un ejemplo
  - mientras tanto, suponga que el algoritmo es correcto
  - después mostraremos que el algoritmo es correcto
- Complejidad de tiempo
  - considere vectores con n elementos
  - ¿cuántas instrucciones son ejecutadas?

Análisis del algoritmo

Qué es importante analizar?

- Correctitud
  - ya probamos el algoritmo con un ejemplo
  - mientras tanto, suponga que el algoritmo es correcto
  - después mostraremos que el algoritmo es correcto
- Complejidad de tiempo
  - considere vectores con n elementos
  - ¿cuántas instrucciones son ejecutadas?

Análisis del algoritmo

Qué es importante analizar?

- Correctitud
  - ya probamos el algoritmo con un ejemplo
  - mientras tanto, suponga que el algoritmo es correcto
  - después mostraremos que el algoritmo es correcto

- Complejidad de tiempo
  - considere vectores con n elementos
  - ¿cuántas instrucciones son ejecutadas?

Análisis del algoritmo

Qué es importante analizar?

- Correctitud
  - ya probamos el algoritmo con un ejemplo
  - mientras tanto, suponga que el algoritmo es correcto
  - después mostraremos que el algoritmo es correcto

- Complejidad de tiempo
  - considere vectores con n elementos
  - ¿cuántas instrucciones son ejecutadas?

Análisis del algoritmo

Qué es importante analizar?

- Correctitud
  - ya probamos el algoritmo con un ejemplo
  - mientras tanto, suponga que el algoritmo es correcto
  - después mostraremos que el algoritmo es correcto
- Complejidad de tiempo
  - considere vectores con n elementos
  - ¿cuántas instrucciones son ejecutadas?

Análisis del algoritmo

Qué es importante analizar?

- Correctitud
  - ya probamos el algoritmo con un ejemplo
  - mientras tanto, suponga que el algoritmo es correcto
  - después mostraremos que el algoritmo es correcto
- Complejidad de tiempo
  - considere vectores con n elementos
  - ¿cuántas instrucciones son ejecutadas?

Análisis del algoritmo

Qué es importante analizar?

- Correctitud
  - ya probamos el algoritmo con un ejemplo
  - mientras tanto, suponga que el algoritmo es correcto
  - después mostraremos que el algoritmo es correcto
- Complejidad de tiempo
  - considere vectores con n elementos
  - ¿cuántas instrucciones son ejecutadas?

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	?	?
2 $\quad llave \leftarrow A[j]$	?	?
3 $\quad i \leftarrow j-1$	?	?
4 $\quad$ **mientras** $i \geq 1$ **y** $A[i] > llave$ **hacer**	?	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	?	?
6 $\quad\quad i \leftarrow i-1$	?	?
7 $\quad A[i+1] \leftarrow llave$	?	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces mientras se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	?
2 $\quad$ *llave* $\leftarrow A[j]$	c_2	?
3 $\quad$ $i \leftarrow j-1$	c_3	?
4 $\quad$ **mientras** $i \geq 1$ **y** $A[i] >$ *llave* **hacer**	c_4	?
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad$ $i \leftarrow i-1$	c_6	?
7 $\quad$ $A[i+1] \leftarrow$ *llave*	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces mientras se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	?
2 $\quad llave \leftarrow A[j]$	c_2	?
3 $\quad i \leftarrow j-1$	c_3	?
4 $\quad$ **mientras** $i \geq 1$ **y** $A[i] > llave$ **hacer**	c_4	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow llave$	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces mientras se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	n
2 $\quad$ *llave* $\leftarrow A[j]$	c_2	?
3 $\quad$ $i \leftarrow j-1$	c_3	?
4 $\quad$ **mientras** $i \geq 1$ y $A[i] >$ *llave* **hacer**	c_4	?
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad$ $i \leftarrow i-1$	c_6	?
7 $\quad$ $A[i+1] \leftarrow$ *llave*	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces mientras se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	n
2 $\quad llave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	?
4 $\quad$ **mientras** $i \geq 1$ **y** $A[i] > llave$ **hacer**	c_4	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow llave$	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces mientras se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	n
2 $\quad llave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **mientras** $i \geq 1$ **y** $A[i] > llave$ **hacer**	c_4	?
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow llave$	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces mientras se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	n
2 $\quad$ *llave* $\leftarrow A[j]$	c_2	$n-1$
3 $\quad$ $i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **mientras** $i \geq 1$ y $A[i] >$ *llave* **hacer**	c_4	?
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad$ $i \leftarrow i-1$	c_6	?
7 $\quad$ $A[i+1] \leftarrow$ *llave*	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces **mientras** se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	n
2 $\quad llave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **mientras** $i \geq 1$ y $A[i] > llave$ **hacer**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	?
6 $\quad\quad i \leftarrow i-1$	c_6	?
7 $\quad A[i+1] \leftarrow llave$	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces **mientras** se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	n
2 $\quad$ *llave* $\leftarrow A[j]$	c_2	$n-1$
3 $\quad$ $i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **mientras** $i \geq 1$ **y** $A[i] >$ *llave* **hacer**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6 $\quad\quad$ $i \leftarrow i-1$	c_6	?
7 $\quad$ $A[i+1] \leftarrow$ *llave*	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces **mientras** se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	n
2 $\quad$ *llave* $\leftarrow A[j]$	c_2	$n-1$
3 $\quad$ $i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **mientras** $i \geq 1$ y $A[i] >$ *llave* **hacer**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad$ $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6 $\quad\quad$ $i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $\quad$ $A[i+1] \leftarrow$ *llave*	c_7	?

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces **mientras** se ejecuta para un cierto j

Contando el número de instrucciones

INSERTION-SORT(A, n)	Costo	¿Cuántas veces?
1 **para** $j \leftarrow 2$ **hasta** n **hacer**	c_1	n
2 $\quad llave \leftarrow A[j]$	c_2	$n-1$
3 $\quad i \leftarrow j-1$	c_3	$n-1$
4 $\quad$ **mientras** $i \geq 1$ y $A[i] > llave$ **hacer**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6 $\quad\quad i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $\quad A[i+1] \leftarrow llave$	c_7	$n-1$

- la línea k ejecuta un número constante de instrucciones c_k
- cada línea se ejecuta una o mas veces
- cuantas veces la línea 4 ejecuta es algo que depende de la entrada
  - sea t_j cuantas veces **mientras** se ejecuta para un cierto j

Tiempo de ejecución total

- Considere una instancia de tamaño n
- Sea $T(n)$ el número de instrucciones ejecutadas para ella
- Basta sumar para todas las líneas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observaciones:

- las entradas del mismo tamaño tienen tiempos diferentes
- vamos a considerar diferentes instancias
  - mejor caso: cuando $T(n)$ es el menor posible
  - peor caso: cuando $T(n)$ es el mayor posible

Tiempo de ejecución total

- Considere una instancia de tamaño n
- Sea $T(n)$ el número de instrucciones ejecutadas para ella
- Basta sumar para todas las líneas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observaciones:

- las entradas del mismo tamaño tienen tiempos diferentes
- vamos a considerar diferentes instancias
  - mejor caso: cuando $T(n)$ es el menor posible
  - peor caso: cuando $T(n)$ es el mayor posible

Tiempo de ejecución total

- Considere una instancia de tamaño n
- Sea $T(n)$ el número de instrucciones ejecutadas para ella
- Basta sumar para todas las líneas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observaciones:

- las entradas del mismo tamaño tienen tiempos diferentes
- vamos a considerar diferentes instancias
  - mejor caso: cuando $T(n)$ es el menor posible
  - peor caso: cuando $T(n)$ es el mayor posible

Tiempo de ejecución total

- Considere una instancia de tamaño n
- Sea $T(n)$ el número de instrucciones ejecutadas para ella
- Basta sumar para todas las líneas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observaciones:

- las entradas del mismo tamaño tienen tiempos diferentes
- vamos a considerar diferentes instancias
  - mejor caso: cuando $T(n)$ es el menor posible
  - peor caso: cuando $T(n)$ es el mayor posible

Tiempo de ejecución total

- Considere una instancia de tamaño n
- Sea $T(n)$ el número de instrucciones ejecutadas para ella
- Basta sumar para todas las líneas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observaciones:

- las entradas del mismo tamaño tienen tiempos diferentes
- vamos a considerar diferentes instancias
  - mejor caso: cuando $T(n)$ es el menor posible
  - peor caso: cuando $T(n)$ es el mayor posible

Tiempo de ejecución total

- Considere una instancia de tamaño n
- Sea $T(n)$ el número de instrucciones ejecutadas para ella
- Basta sumar para todas las líneas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observaciones:

- las entradas del mismo tamaño tienen tiempos diferentes
- vamos a considerar diferentes instancias
  - mejor caso: cuando $T(n)$ es el menor posible
  - peor caso: cuando $T(n)$ es el mayor posible

Tiempo de ejecución total

- Considere una instancia de tamaño n
- Sea $T(n)$ el número de instrucciones ejecutadas para ella
- Basta sumar para todas las líneas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observaciones:

- las entradas del mismo tamaño tienen tiempos diferentes
- vamos a considerar diferentes instancias
  - mejor caso: cuando $T(n)$ es el menor posible
  - peor caso: cuando $T(n)$ es el mayor posible

Tiempo de ejecución total

- Considere una instancia de tamaño n
- Sea $T(n)$ el número de instrucciones ejecutadas para ella
- Basta sumar para todas las líneas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observaciones:

- las entradas del mismo tamaño tienen tiempos diferentes
- vamos a considerar diferentes instancias
  - mejor caso: cuando $T(n)$ es el menor posible
  - peor caso: cuando $T(n)$ es el mayor posible

Mejor caso

Un mejor caso ocurre cuando $t_j = 1$ para cada j

- basta que la condición del **mientras** siempre falle
- ocurre si la entrada A ya está ordenada

En ese caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- los valores de a y b son constantes
- el tiempo de ejecución en el mejor caso es lineal en n

Mejor caso

Un mejor caso ocurre cuando $t_j = 1$ para cada j

- basta que la condición del **mientras** siempre falle
- ocurre si la entrada A ya está ordenada

En ese caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- los valores de a y b son constantes
- el tiempo de ejecución en el mejor caso es lineal en n

Mejor caso

Un mejor caso ocurre cuando $t_j = 1$ para cada j

- basta que la condición del **mientras** siempre falle
- ocurre si la entrada A ya está ordenada

En ese caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- los valores de a y b son constantes
- el tiempo de ejecución en el mejor caso es lineal en n

Mejor caso

Un mejor caso ocurre cuando $t_j = 1$ para cada j

- basta que la condición del **mientras** siempre falle
- ocurre si la entrada A ya está ordenada

En ese caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- los valores de a y b son constantes
- el tiempo de ejecución en el mejor caso es lineal en n

Mejor caso

Un mejor caso ocurre cuando $t_j = 1$ para cada j

- basta que la condición del **mientras** siempre falle
- ocurre si la entrada A ya está ordenada

En ese caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- los valores de a y b son constantes
- el tiempo de ejecución en el mejor caso es lineal en n

Mejor caso

Un mejor caso ocurre cuando $t_j = 1$ para cada j

- basta que la condición del **mientras** siempre falle
- ocurre si la entrada A ya está ordenada

En ese caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- los valores de a y b son constantes
- el tiempo de ejecución en el mejor caso es lineal en n

Mejor caso

Un mejor caso ocurre cuando $t_j = 1$ para cada j

- basta que la condición del **mientras** siempre falle
- ocurre si la entrada A ya está ordenada

En ese caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- los valores de a y b son constantes
- el tiempo de ejecución en el mejor caso es lineal en n

Peor caso

Un peor caso ocurre cuando t_j es máximo para cada j

- basta que la condición del **mientras** falle cuando $i = 0$
- en esa situación, tendremos que $t_j = j$
- ocurre si la entrada A está ordenada decrecientemente

Recuerde que

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

Peor caso

Un peor caso ocurre cuando t_j es máximo para cada j

- basta que la condición del **mientras** falle cuando $i = 0$
- en esa situación, tendremos que $t_j = j$
- ocurre si la entrada A está ordenada decrecientemente

Recuerde que

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

Peor caso

Un peor caso ocurre cuando t_j es máximo para cada j

- basta que la condición del **mientras** falle cuando $i = 0$
- en esa situación, tendremos que $t_j = j$
- ocurre si la entrada A está ordenada decrecientemente

Recuerde que

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

Peor caso

Un peor caso ocurre cuando t_j es máximo para cada j

- basta que la condición del **mientras** falle cuando $i = 0$
- en esa situación, tendremos que $t_j = j$
- ocurre si la entrada A está ordenada decrecientemente

Recuerde que

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

Peor caso

Un peor caso ocurre cuando t_j es máximo para cada j

- basta que la condición del **mientras** falle cuando $i = 0$
- en esa situación, tendremos que $t_j = j$
- ocurre si la entrada A está ordenada decrecientemente

Recuerde que

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

Peor caso

Un peor caso ocurre cuando t_j es máximo para cada j

- basta que la condición del **mientras** falle cuando $i = 0$
- en esa situación, tendremos que $t_j = j$
- ocurre si la entrada A está ordenada decrecientemente

Recuerde que

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

Peor caso (cont)

Sustituyendo, tenemos

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n(n+1)/2 - 1) + \\ &\quad c_5 \cdot n(n-1)/2 + c_6 \cdot n(n-1)/2 + c_7 \cdot (n-1) \\ &= (c_4/2 + c_5/2 + c_6/2) \cdot n^2 + \\ &\quad (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7) \cdot n - \\ &\quad (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n^2 + b \cdot n + c \end{aligned}$$

- los valores de a, b, c son constantes
- el tiempo de ejecución en el peor caso es cuadrático en n

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Complejidad asintótica

Estamos interesados principalmente

- en el análisis del peor caso
- en el tiempo de ejecución para instancias grandes

Comportamiento asintótico

- en el peor caso tenemos $T(n) = an^2 + bn + c$
  - el término dominante es el que contiene n^2
  - el tiempo de ejecución es una función cuadrática
  - las constantes a, b, c sólo dependen de la implementación
- no nos preocupamos con los valores de a, b, c

¿Porqué esto es razonable?

Um ejemplo de función cuadrática

Considere la función $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- cuando n es grande, el término $3n^2$ es una buena estimativa
- podemos concentrarnos en los términos dominantes

Um ejemplo de función cuadrática

Considere la función $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- cuando n es grande, el término $3n^2$ es una buena estimativa
- podemos concentrarnos en los términos dominantes

Um ejemplo de función cuadrática

Considere la función $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- cuando n es grande, el término $3n^2$ es una buena estimativa
- podemos concentrarnos en los términos dominantes

Um ejemplo de función cuadrática

Considere la función $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- cuando n es grande, el término $3n^2$ es una buena estimativa
- podemos concentrarnos en los términos dominantes

Notación asintótica

¿Cómo simplificar el tiempo del peor caso de INSERTION-SORT?

- en lugar de escribir $T(n) = an^2 + bn + c$,
- escribimos solamente $T(n) = \Theta(n^2)$

Esa notación significa que, para n suficientemente grande,

1. $T(n)$ es limitada superiormente por $c \cdot n^2$, para algún $c > 0$
2. $T(n)$ es limitada inferiormente por $d \cdot n^2$, para algún $d > 0$

Posteriormente formalizaremos esa notación assintótica!

Notación asintótica

¿Cómo simplificar el tiempo del peor caso de Insertion-Sort?

- en lugar de escribir $T(n) = an^2 + bn + c$,
- escribimos solamente $T(n) = \Theta(n^2)$

Esa notación significa que, para n suficientemente grande,

1. $T(n)$ es limitada superiormente por $c \cdot n^2$, para algún $c > 0$
2. $T(n)$ es limitada inferiormente por $d \cdot n^2$, para algún $d > 0$

Posteriormente formalizaremos esa notación assintótica!

Notación asintótica

¿Cómo simplificar el tiempo del peor caso de INSERTION-SORT?

- en lugar de escribir $T(n) = an^2 + bn + c$,
- escribimos solamente $T(n) = \Theta(n^2)$

Esa notación significa que, para n suficientemente grande,

1. $T(n)$ es limitada superiormente por $c \cdot n^2$, para algún $c > 0$
2. $T(n)$ es limitada inferiormente por $d \cdot n^2$, para algún $d > 0$

Posteriormente formalizaremos esa notación assintótica!

Notación asintótica

¿Cómo simplificar el tiempo del peor caso de INSERTION-SORT?

- en lugar de escribir $T(n) = an^2 + bn + c$,
- escribimos solamente $T(n) = \Theta(n^2)$

Esa notación significa que, para n suficientemente grande,

1. $T(n)$ es limitada superiormente por $c \cdot n^2$, para algún $c > 0$
2. $T(n)$ es limitada inferiormente por $d \cdot n^2$, para algún $d > 0$

Posteriormente formalizaremos esa notación assintótica!

Notación asintótica

¿Cómo simplificar el tiempo del peor caso de INSERTION-SORT?

- en lugar de escribir $T(n) = an^2 + bn + c$,
- escribimos solamente $T(n) = \Theta(n^2)$

Esa notación significa que, para n suficientemente grande,

1. $T(n)$ es limitada superiormente por $c \cdot n^2$, para algún $c > 0$
2. $T(n)$ es limitada inferiormente por $d \cdot n^2$, para algún $d > 0$

Posteriormente formalizaremos esa notación assintótica!

Notación asintótica

¿Cómo simplificar el tiempo del peor caso de Insertion-Sort?

- en lugar de escribir $T(n) = an^2 + bn + c$,
- escribimos solamente $T(n) = \Theta(n^2)$

Esa notación significa que, para n suficientemente grande,

1. $T(n)$ es limitada superiormente por $c \cdot n^2$, para algún $c > 0$
2. $T(n)$ es limitada inferiormente por $d \cdot n^2$, para algún $d > 0$

Posteriormente formalizaremos esa notación assintótica!

Notación asintótica

¿Cómo simplificar el tiempo del peor caso de INSERTION-SORT?

- en lugar de escribir $T(n) = an^2 + bn + c$,
- escribimos solamente $T(n) = \Theta(n^2)$

Esa notación significa que, para n suficientemente grande,

1. $T(n)$ es limitada superiormente por $c \cdot n^2$, para algún $c > 0$
2. $T(n)$ es limitada inferiormente por $d \cdot n^2$, para algún $d > 0$

Posteriormente formalizaremos esa notación assintótica!

Proyectando Algoritmos

Proyectando Algoritmos

- Divide y Venceras

Entendiendo y Mejorando

Hasta ahora:

- ordenamos incrementalmente con Insertion-Sort
- vimos que su complejidad para el peor caso es $\Theta(n^2)$

Vamos a estudiar una manera alternativa de ordenar números

- vamos a utilizar una técnica recursiva llamada divide y vencerás
- muchas veces, obtenemos algoritmos mas rápidos que los incrementales

Entendiendo y Mejorando

Hasta ahora:

- ordenamos incrementalmente con Insertion-Sort
- vimos que su complejidad para el peor caso es $\Theta(n^2)$

Vamos a estudiar una manera alternativa de ordenar números

- vamos a utilizar una técnica recursiva llamada divide y vencerás
- muchas veces, obtenemos algoritmos mas rápidos que los incrementales

Entendiendo y Mejorando

Hasta ahora:

- ordenamos incrementalmente con Insertion-Sort
- vimos que su complejidad para el peor caso es $\Theta(n^2)$

Vamos a estudiar una manera alternativa de ordenar números

- vamos a utilizar una técnica recursiva llamada divide y vencerás
- muchas veces, obtenemos algoritmos mas rápidos que los incrementales

Entendiendo y Mejorando

Hasta ahora:

- ordenamos incrementalmente con Insertion-Sort
- vimos que su complejidad para el peor caso es $\Theta(n^2)$

Vamos a estudiar una manera alternativa de ordenar números

- vamos a utilizar una técnica recursiva llamada divide y vencerás
- muchas veces, obtenemos algoritmos mas rápidos que los incrementales

Entendiendo y Mejorando

Hasta ahora:

- ordenamos incrementalmente con Insertion-Sort
- vimos que su complejidad para el peor caso es $\Theta(n^2)$

Vamos a estudiar una manera alternativa de ordenar números

- vamos a utilizar una técnica recursiva llamada divide y vencerás
- muchas veces, obtenemos algoritmos mas rápidos que los incrementales

Entendiendo y Mejorando

Hasta ahora:

- ordenamos incrementalmente con Insertion-Sort
- vimos que su complejidad para el peor caso es $\Theta(n^2)$

Vamos a estudiar una manera alternativa de ordenar números

- vamos a utilizar una técnica recursiva llamada divide y vencerás
- muchas veces, obtenemos algoritmos mas rápidos que los incrementales

Algoritmos recursivos

> *"To understand recursion, we must first understand recursion."*
> (autor desconhecido)

- Un algoritmo recursivo resuelve un problema
  - directamente, si la instancia fuese pequeña
  - ejecutandose a sí mesmo, si la instancia no fuese pequeña
- La llamada recursiva debe reciber una instancia menor

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(autor desconhecido)

- Un algoritmo recursivo resuelve un problema
  - directamente, si la instancia fuese pequeña
  - ejecutandose a sí mesmo, si la instancia no fuese pequeña
- La llamada recursiva debe reciber una instancia menor

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(autor desconhecido)

- Un algoritmo recursivo resuelve un problema
  - directamente, si la instancia fuese pequeña
  - ejecutandose a sí mesmo, si la instancia no fuese pequeña
- La llamada recursiva debe reciber una instancia menor

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(autor desconhecido)

- Un algoritmo recursivo resuelve un problema
  - directamente, si la instancia fuese pequeña
  - ejecutandose a sí mesmo, si la instancia no fuese pequeña
- La llamada recursiva debe reciber una instancia menor

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(autor desconhecido)

- Un algoritmo recursivo resuelve un problema
  - directamente, si la instancia fuese pequeña
  - ejecutandose a sí mesmo, si la instancia no fuese pequeña
- La llamada recursiva debe reciber una instancia menor

Divide y Vencerás

Un algoritmo de divisão e conquista tiene tres etapas:

1. División: dividir el problema en subproblemas semejantes, pero con instancias menores
2. Conquista: cada subproblema es resuelto recursivamente, o directamente si los subproblemas fuesen pequeños
3. Combinación: las soluciones de los subproblemas son combinadas para obtener uma solución de la instancia original

Divide y Vencerás

Un algoritmo de divisão e conquista tiene tres etapas:

1. División: dividir el problema en subproblemas semejantes, pero con instancias menores
2. Conquista: cada subproblema es resuelto recursivamente, o directamente si los subproblemas fuesen pequeños
3. Combinación: las soluciones de los subproblemas son combinadas para obtener uma solución de la instancia original

Divide y Vencerás

Un algoritmo de divisão e conquista tiene tres etapas:

1. División: dividir el problema en subproblemas semejantes, pero con instancias menores
2. Conquista: cada subproblema es resuelto recursivamente, o directamente si los subproblemas fuesen pequeños
3. Combinación: las soluciones de los subproblemas son combinadas para obtener uma solución de la instancia original

Divide y Vencerás

Un algoritmo de divisão e conquista tiene tres etapas:

1. División: dividir el problema en subproblemas semejantes, pero con instancias menores
2. Conquista: cada subproblema es resuelto recursivamente, o directamente si los subproblemas fuesen pequeños
3. Combinación: las soluciones de los subproblemas son combinadas para obtener uma solución de la instancia original

Ejemplo: ordenando usando divide y vencerás

MERGE-SORT es um ejemplo clásico de divide y vencerás.

Idea:

1. División: divide un vector de tamaño n en dos subvectores de tamaños $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil$
2. Conquista: ordene los dos subvectores recursivamente
3. Combinación: intercala los dos subvectores obteniendo un vector ordenado

Ejemplo: ordenando usando divide y vencerás

MERGE-SORT es um ejemplo clásico de divide y vencerás.

Idea:

1. División: divide un vector de tamaño n en dos subvectores de tamaños $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil$
2. Conquista: ordene los dos subvectores recursivamente
3. Combinación: intercala los dos subvectores obteniendo un vector ordenado

Ejemplo: ordenando usando divide y vencerás

MERGE-SORT es um ejemplo clásico de divide y vencerás.

Idea:

1. División: divide un vector de tamaño n en dos subvectores de tamaños $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil$
2. Conquista: ordene los dos subvectores recursivamente
3. Combinación: intercala los dos subvectores obteniendo un vector ordenado

Ejemplo: ordenando usando divide y vencerás

MERGE-SORT es um ejemplo clásico de divide y vencerás.

Idea:

1. División: divide un vector de tamaño n en dos subvectores de tamaños $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil$
2. Conquista: ordene los dos subvectores recursivamente
3. Combinación: intercala los dos subvectores obteniendo un vector ordenado

Ejemplo: ordenando usando divide y vencerás

MERGE-SORT es um ejemplo clásico de divide y vencerás.

Idea:

1. División: divide un vector de tamaño n en dos subvectores de tamaños $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil$
2. Conquista: ordene los dos subvectores recursivamente
3. Combinación: intercala los dos subvectores obteniendo un vector ordenado

Mergesort

El vector de entrada es representado como $A[p \ldots r]$, con $p \leq r$.

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

	p				q				r
A	66	33	55	44	99	11	77	22	88

Mergesort

El vector de entrada es representado como $A[p \ldots r]$, con $p \leq r$.

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

Mergesort

El vector de entrada es representado como $A[p \ldots r]$, con $p \leq r$.

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

Mergesort

El vector de entrada es representado como $A[p \ldots r]$, con $p \leq r$.

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

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Salida:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Salida:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Salida:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Salida:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Salida:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Salida:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Salida:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

Entrada:

Salida:

Combinando soluciones de los subproblemas

Problema: Intercalar dos subvectores

Entrada: vector $A[p \ldots r]$ tal que

1. subvector $A[p \ldots q]$ está ordenado
2. subvector $A[q+1 \ldots r]$ está ordenado

Salida: vector $A[p \ldots r]$ ordenado

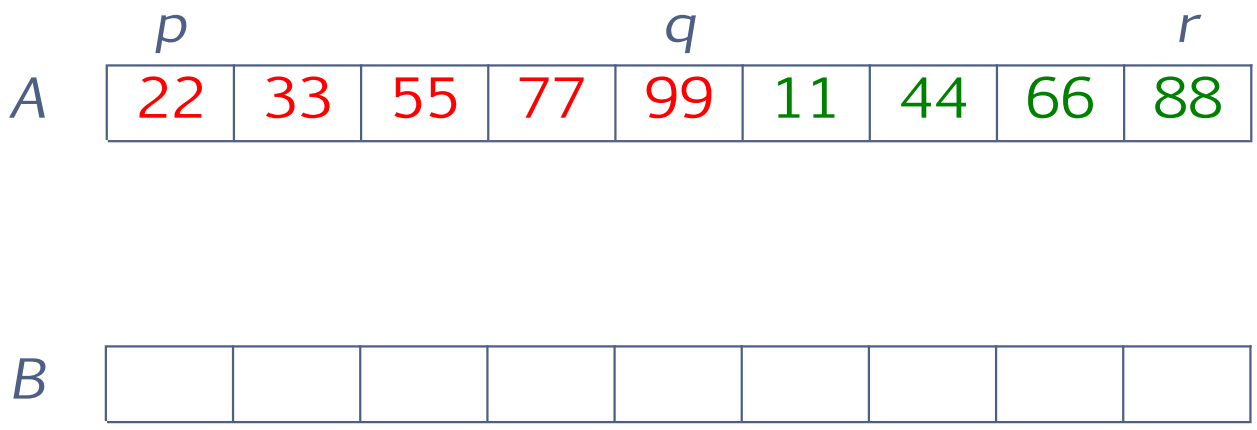
Entrada:

Salida:

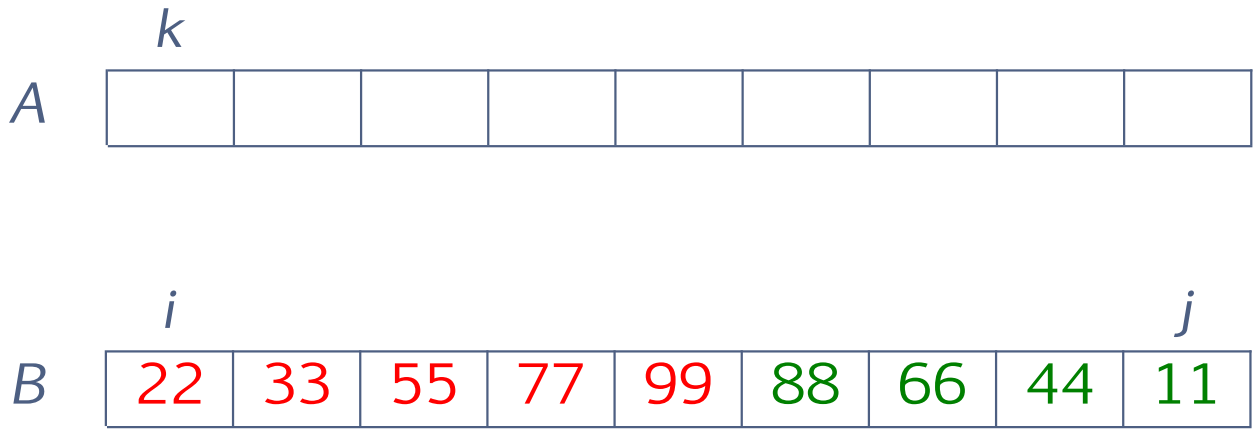
Pseudocódigo de INTERCAMBIA

```
INTERCAMBIA(A, p, q, r)
 1  para i ← p hasta q hacer
 2      B[i] ← A[i]
 3  para j ← q + 1 hasta r hacer
 4      B[r + q + 1 − j] ← A[j]
 5  i ← p
 6  j ← r
 7  para k ← p hasta r hacer
 8      si B[i] ≤ B[j]
 9          entonces A[k] ← B[i]
10                   i ← i + 1
11          sino A[k] ← B[j]
12               j ← j − 1
```

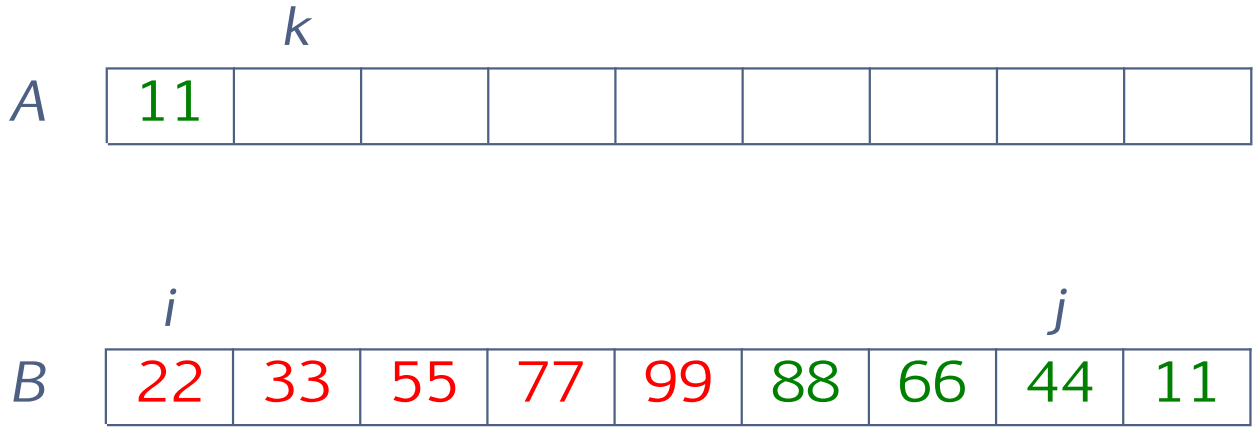
Intercambiando



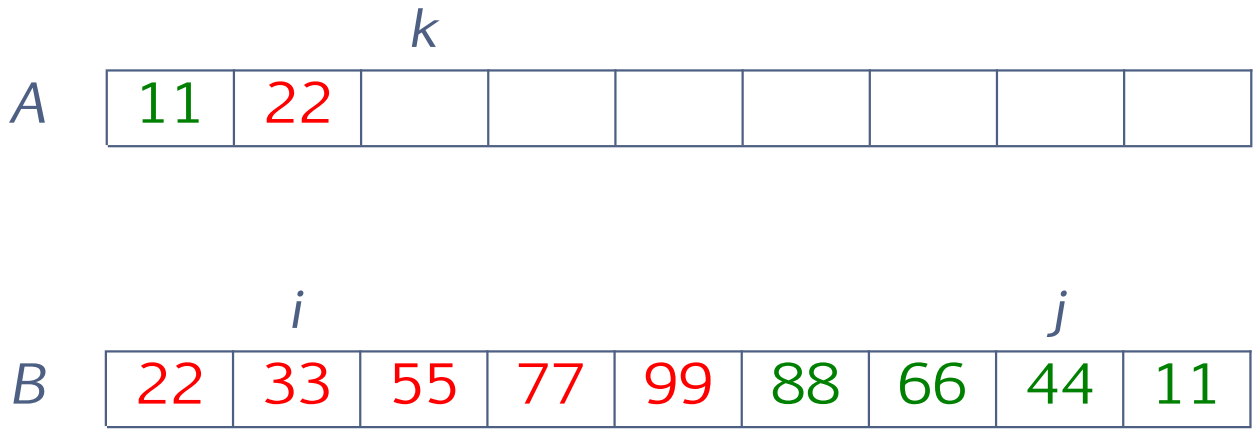
Intercambiando



Intercambiando

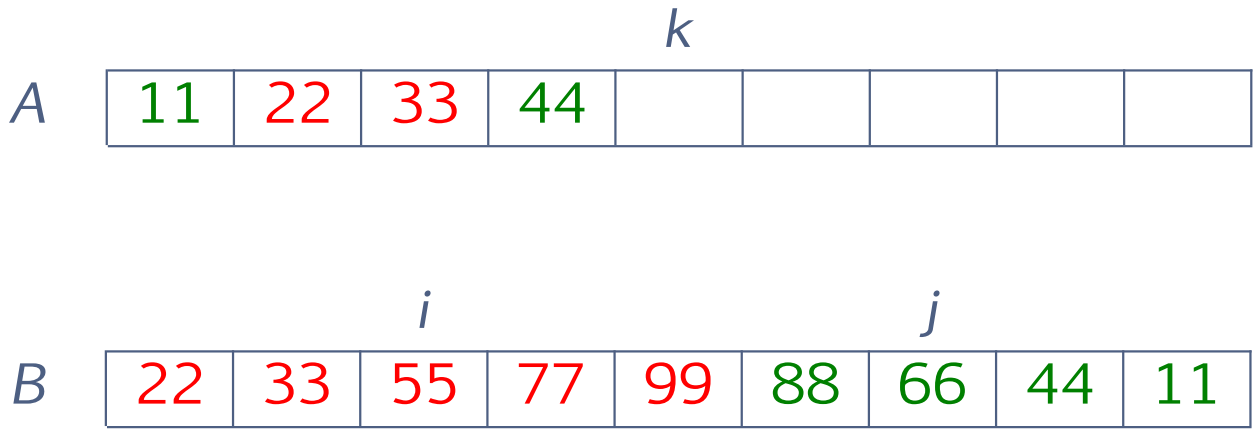


Intercambiando

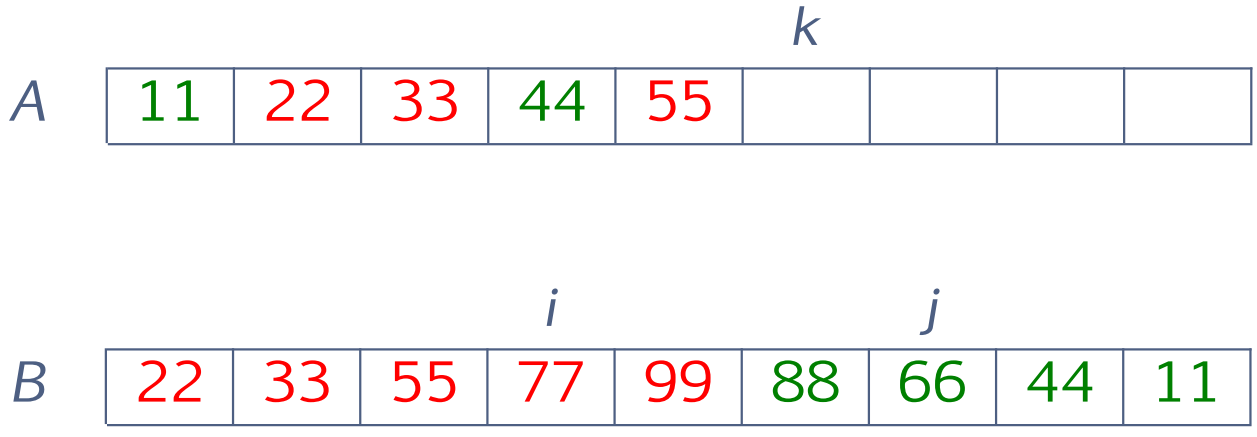


Intercambiando

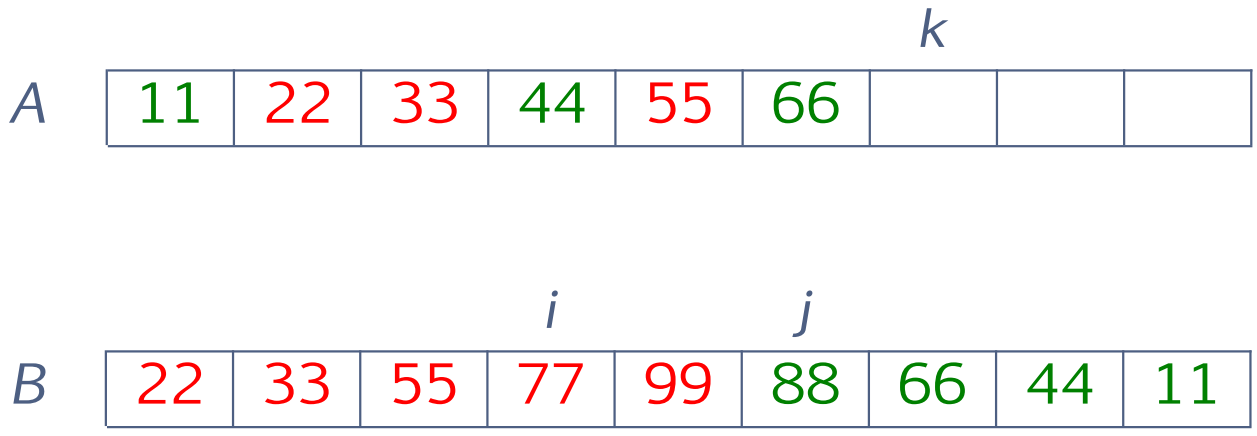
Intercambiando



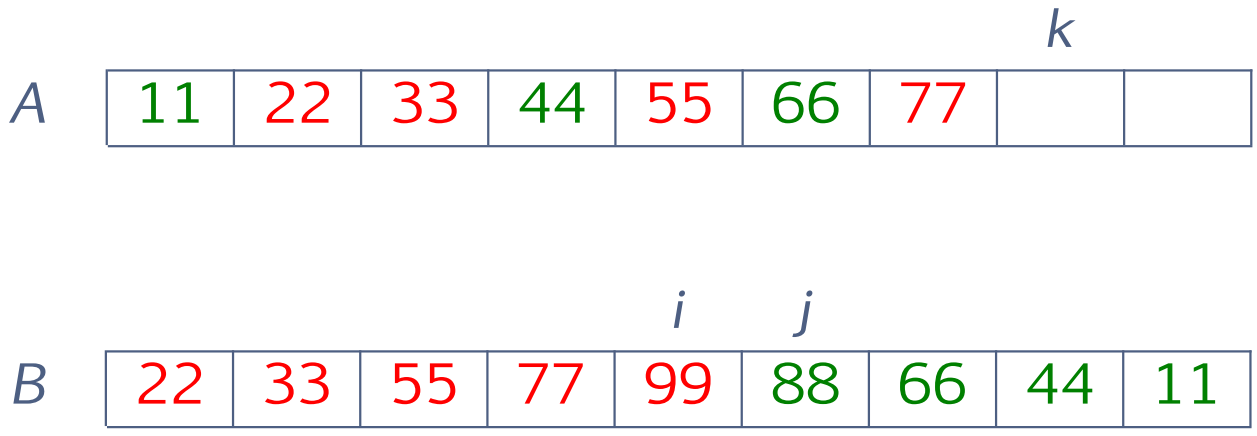
Intercambiando



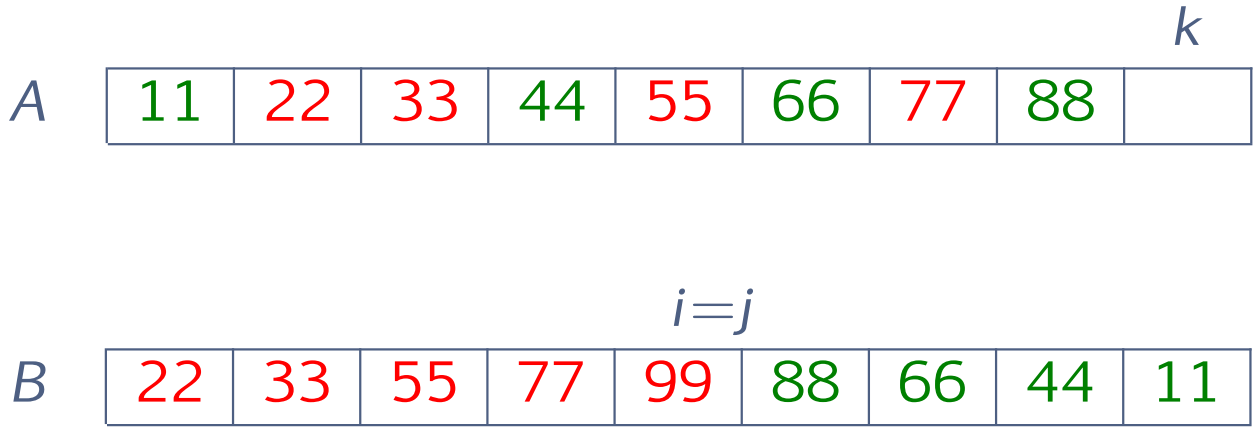
Intercambiando



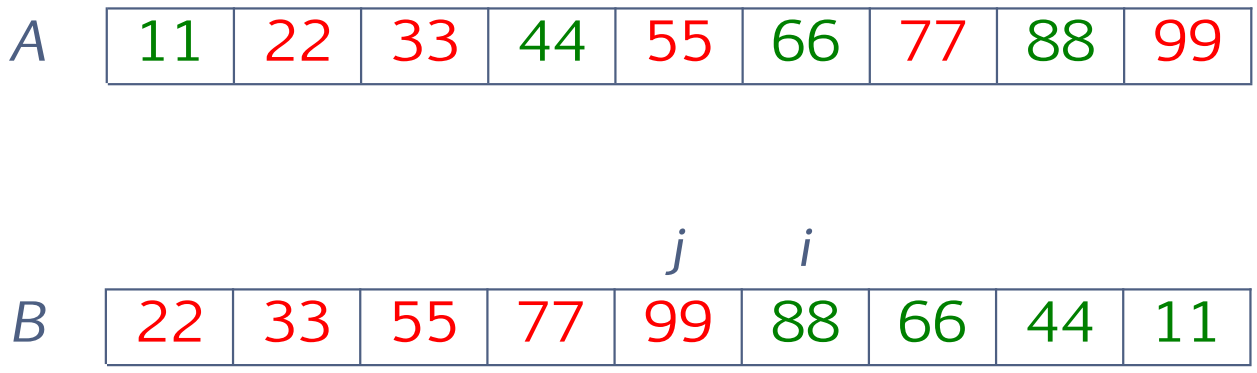
Intercambiando



Intercambiando



Intercambiando



Complejidad de Intercambia

Entrada:

Salida:

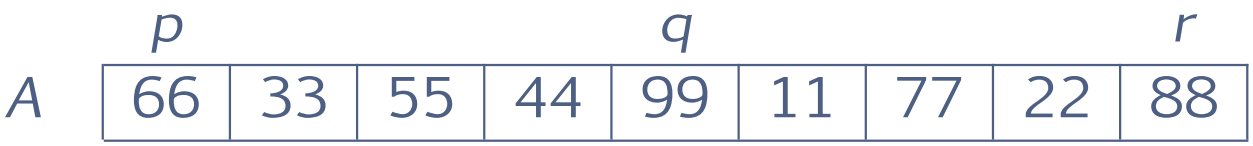
Tamaño da entrada: $n = r - p + 1$

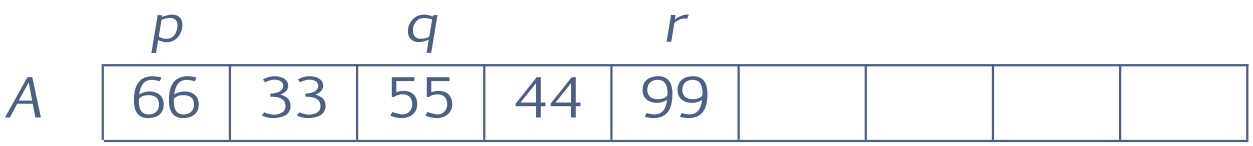
Consumo de tiempo: $\Theta(n)$

Ordenando por intercalación

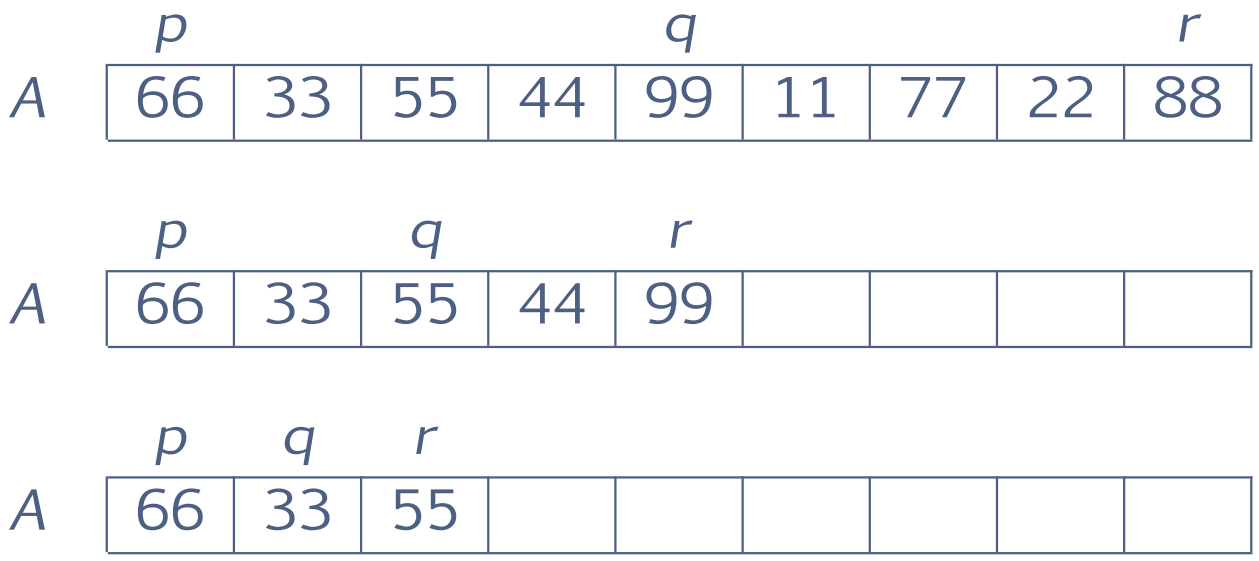
	p				q				r
A	66	33	55	44	99	11	77	22	88

Ordenando por intercalación

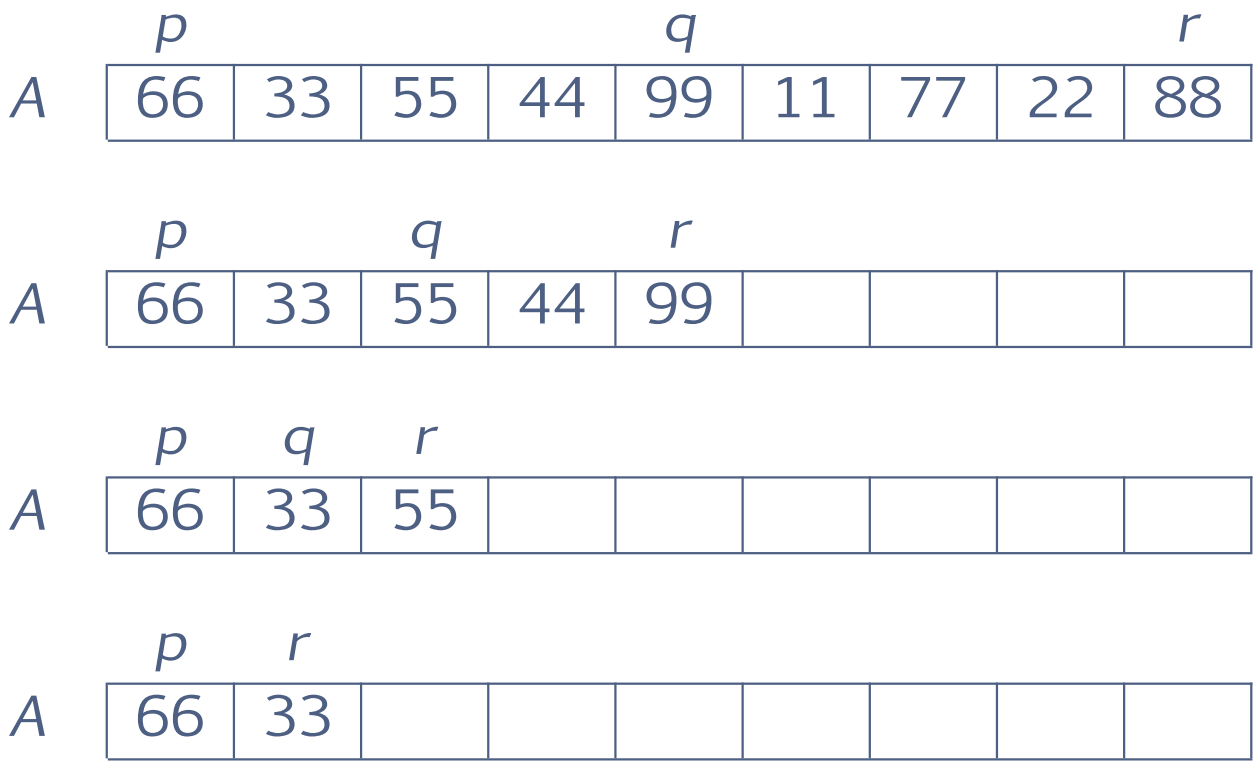




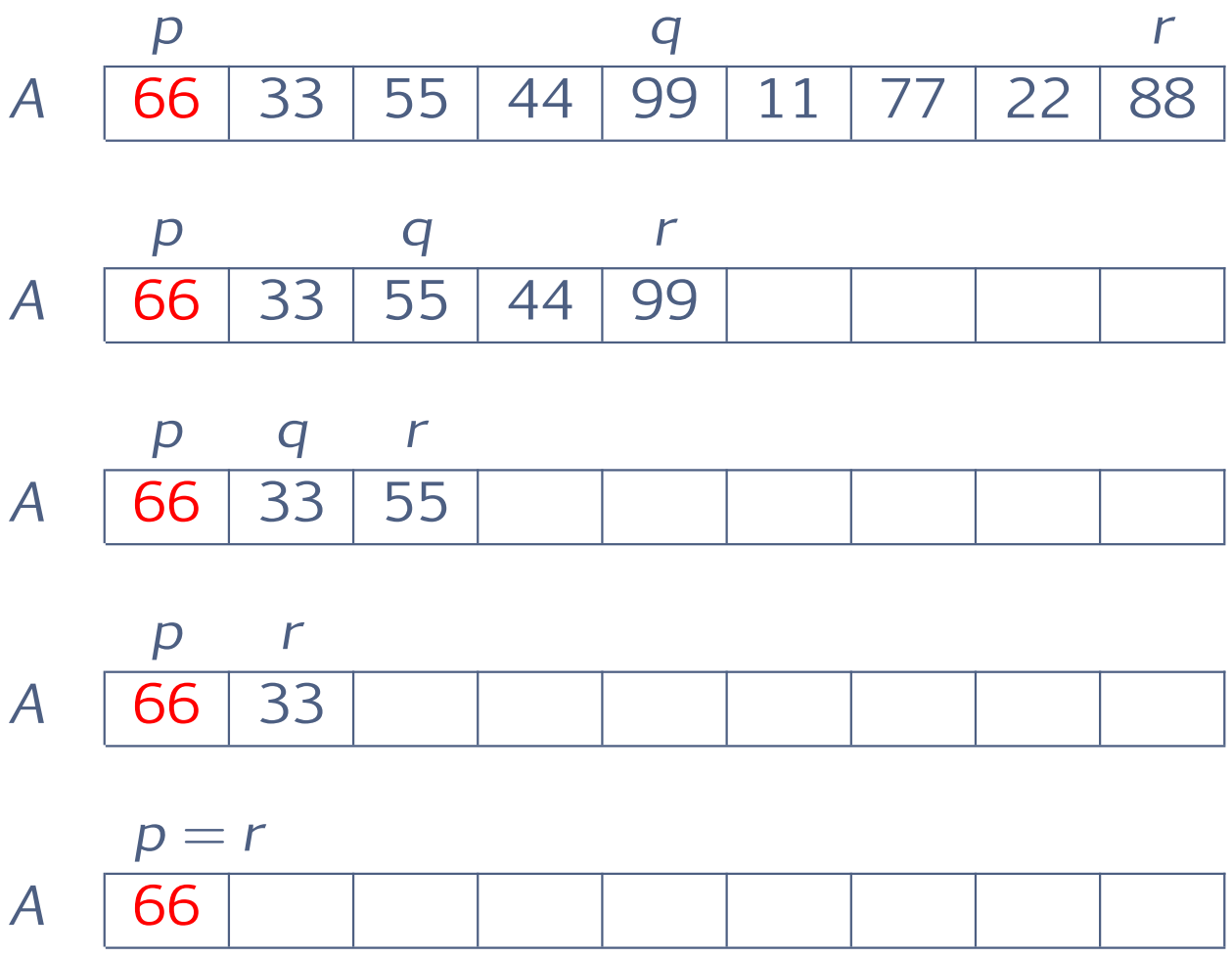
Ordenando por intercalación



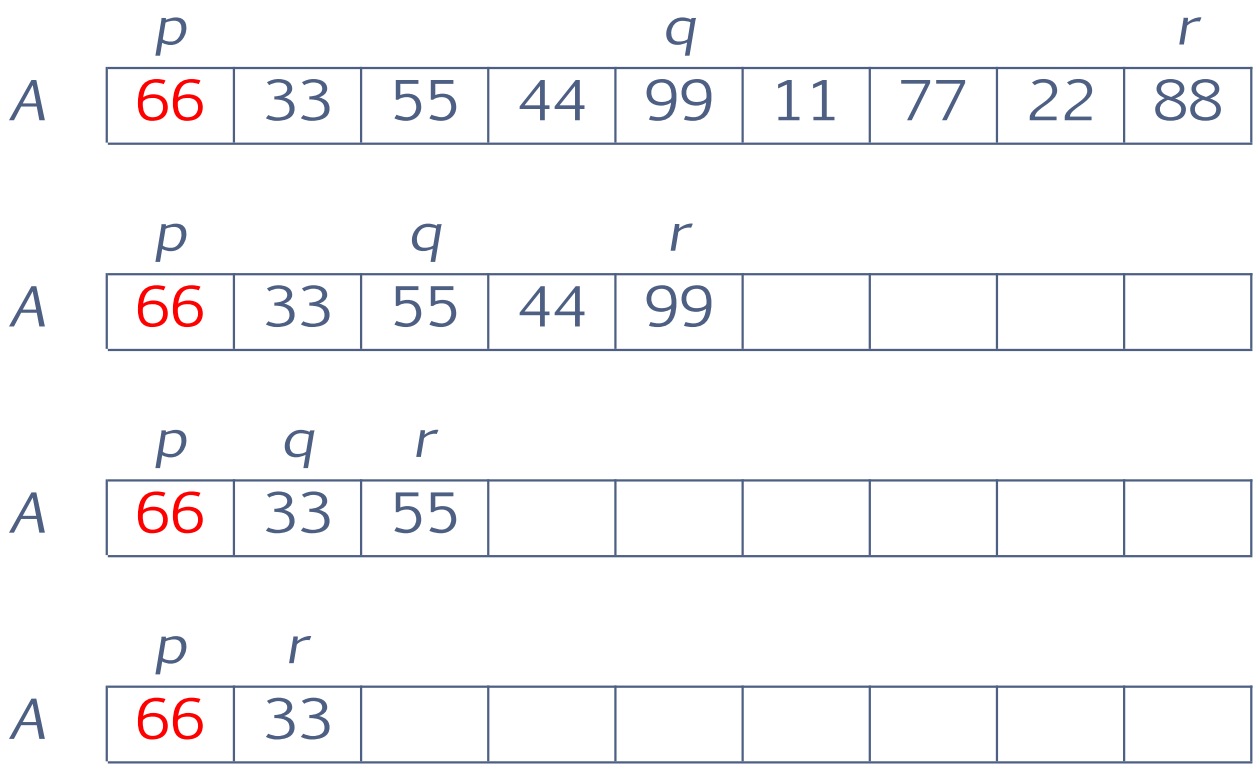
Ordenando por intercalación



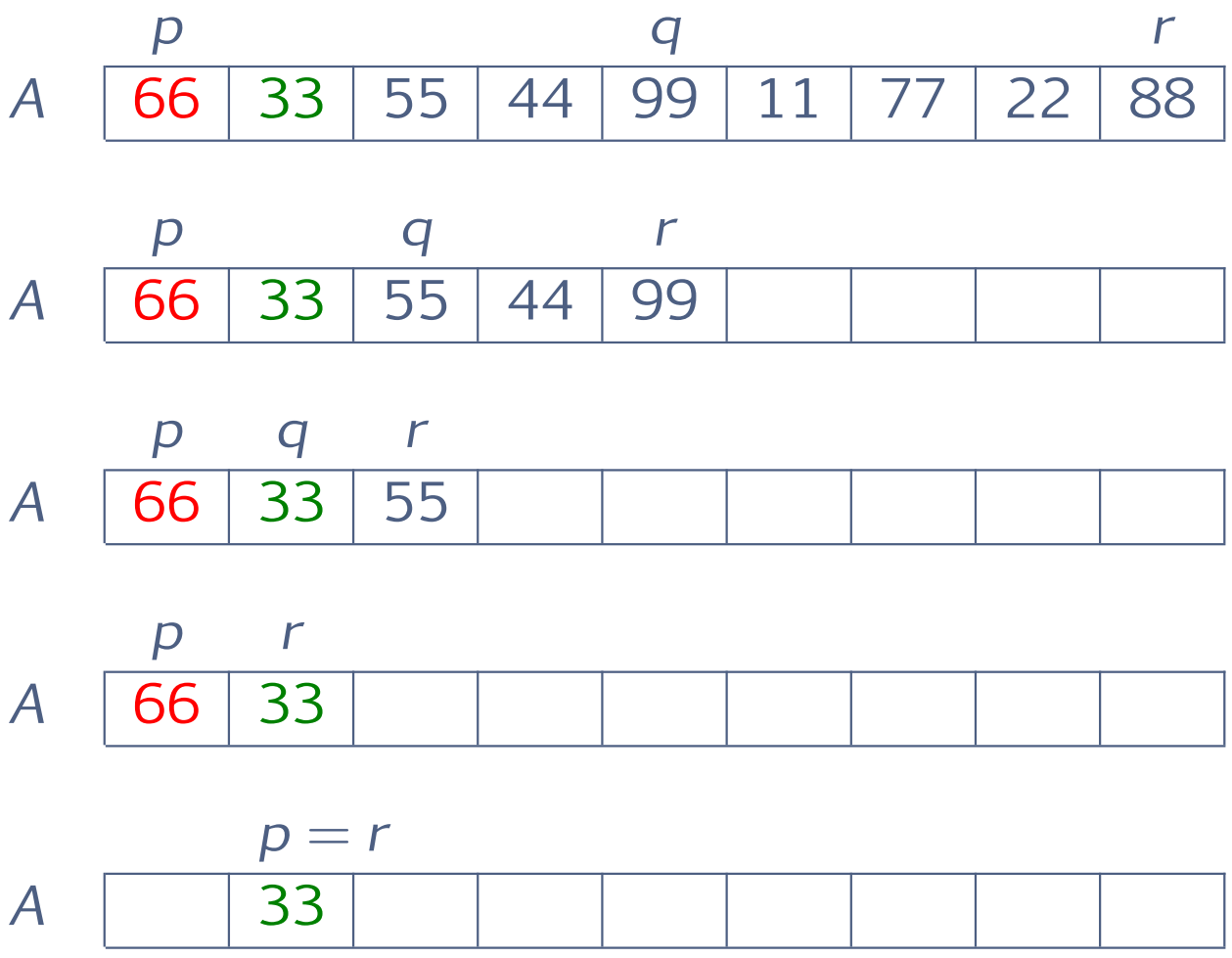
Ordenando por intercalación



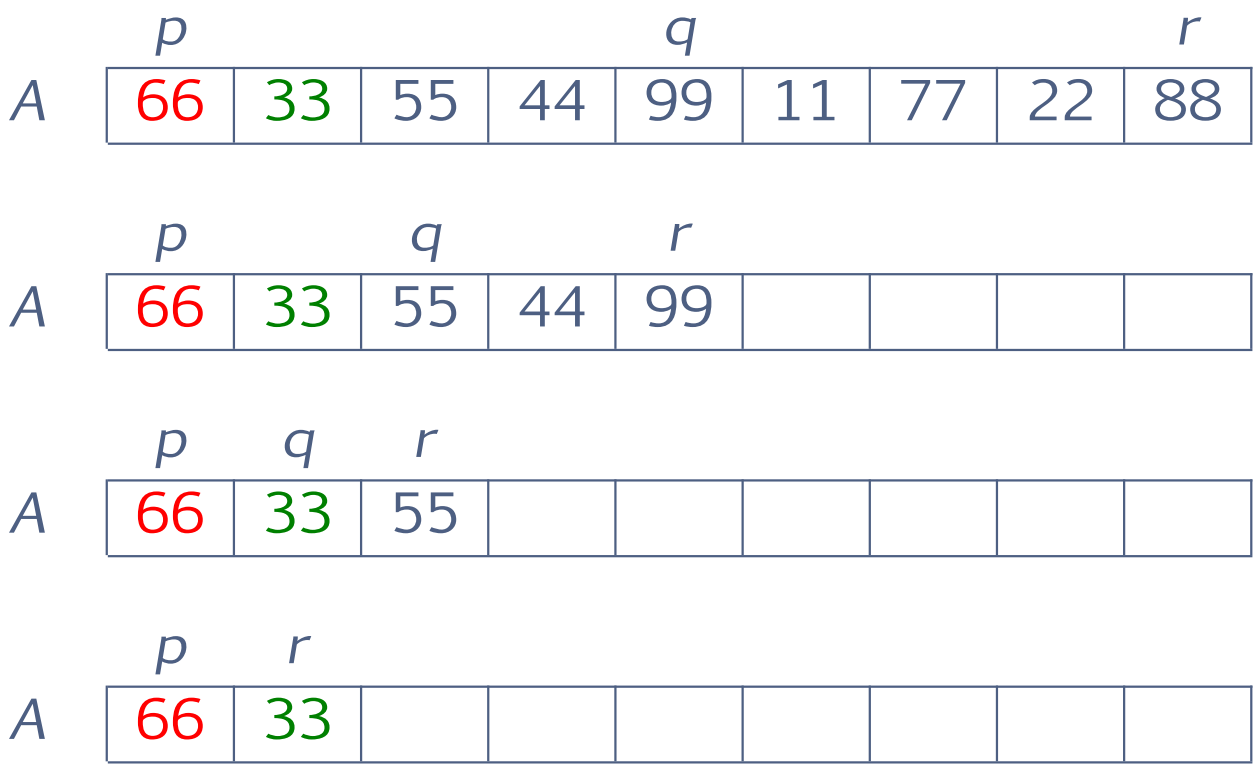
Ordenando por intercalación



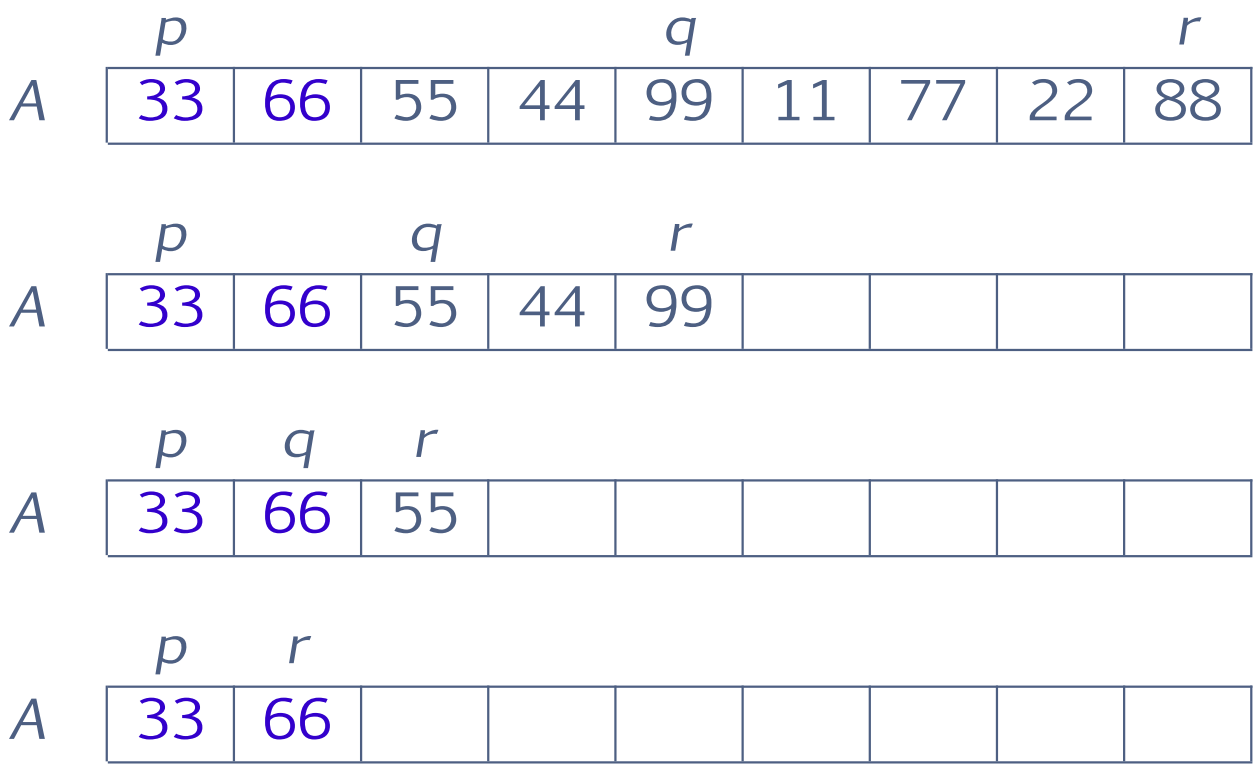
Ordenando por intercalación



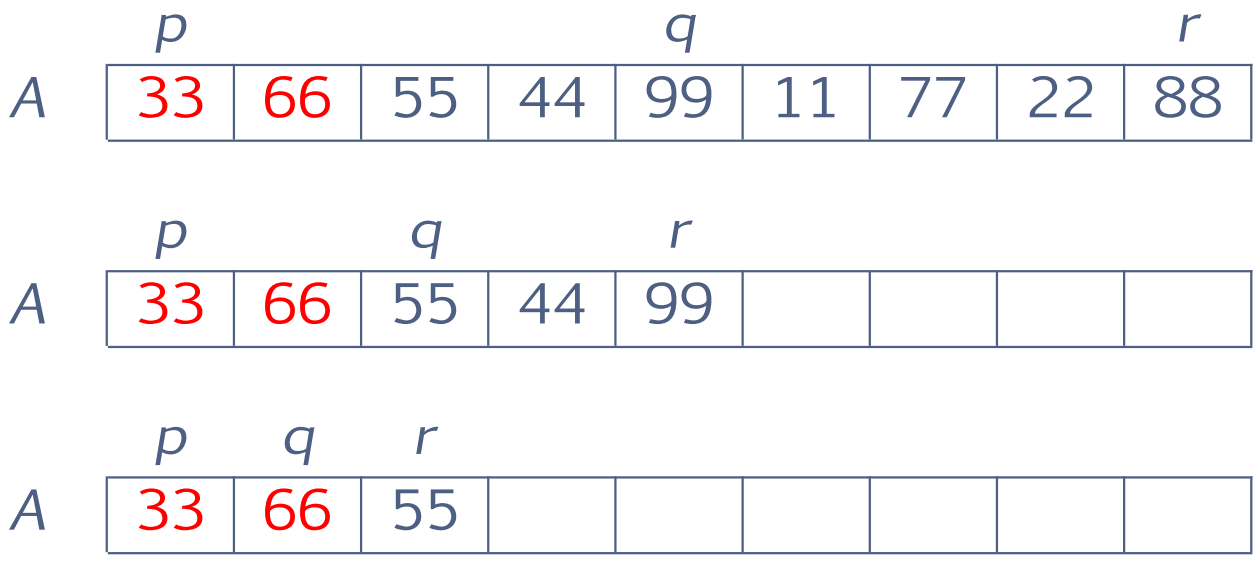
Ordenando por intercalación



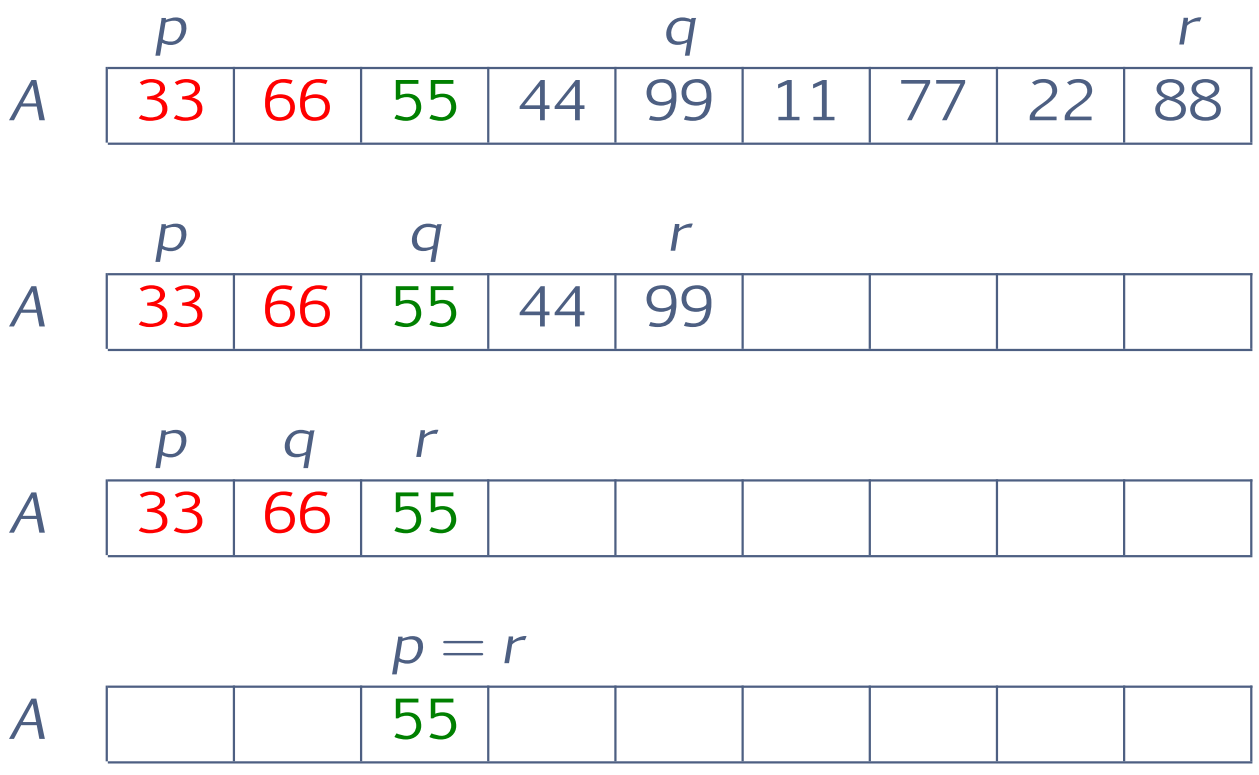
Ordenando por intercalación



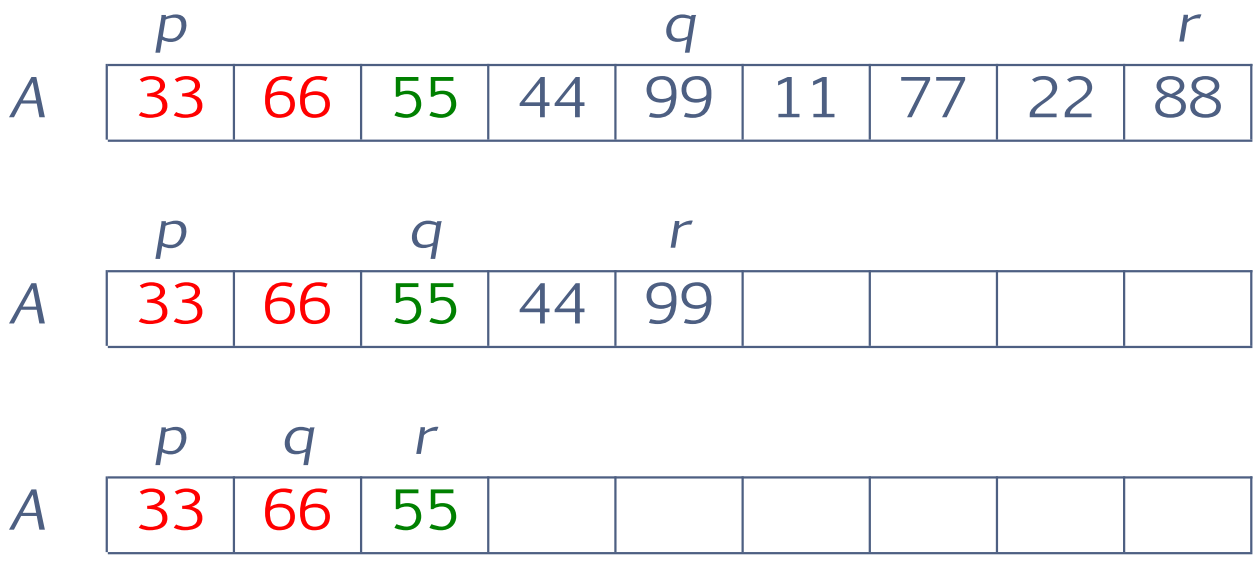
Ordenando por intercalación



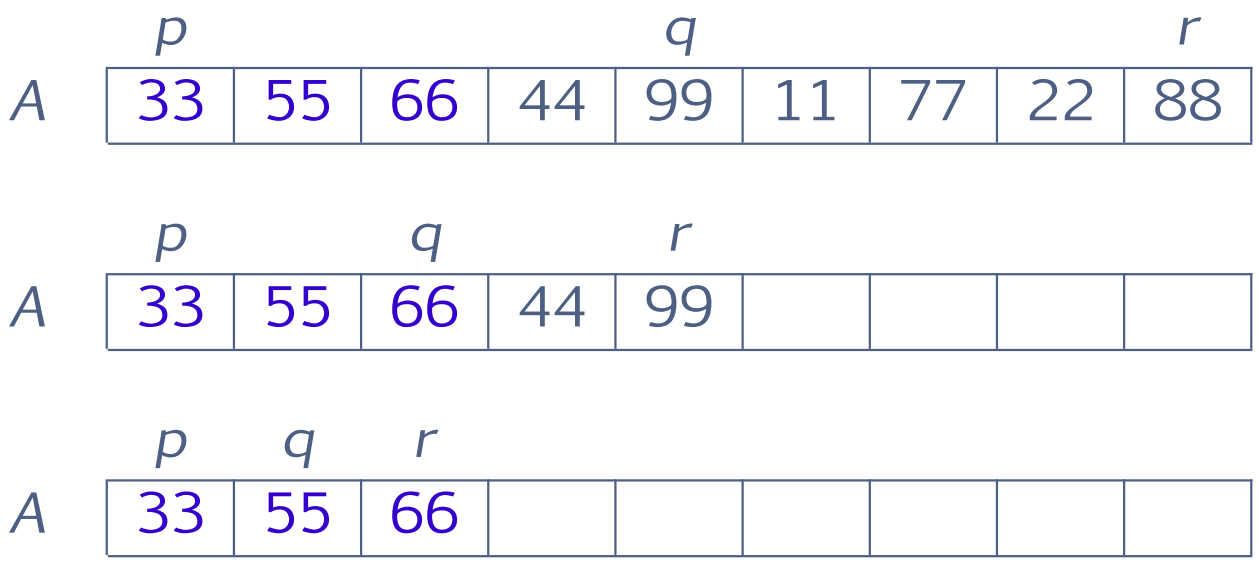
Ordenando por intercalación



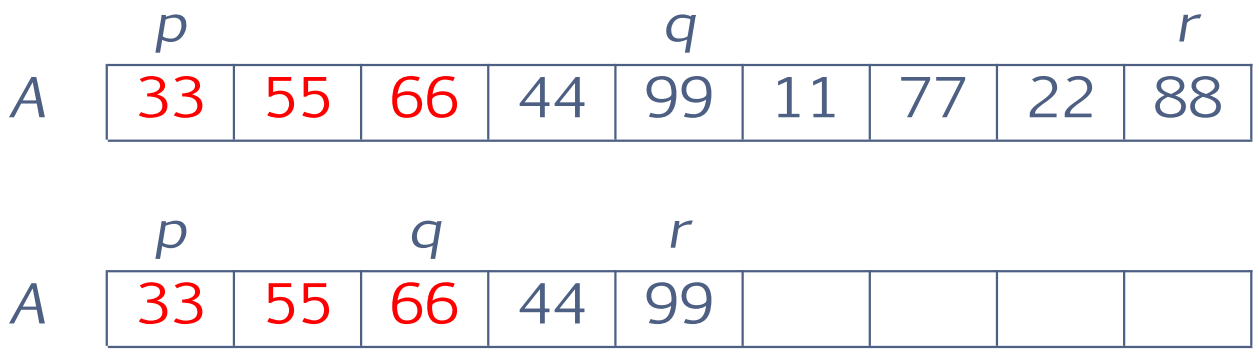
Ordenando por intercalación



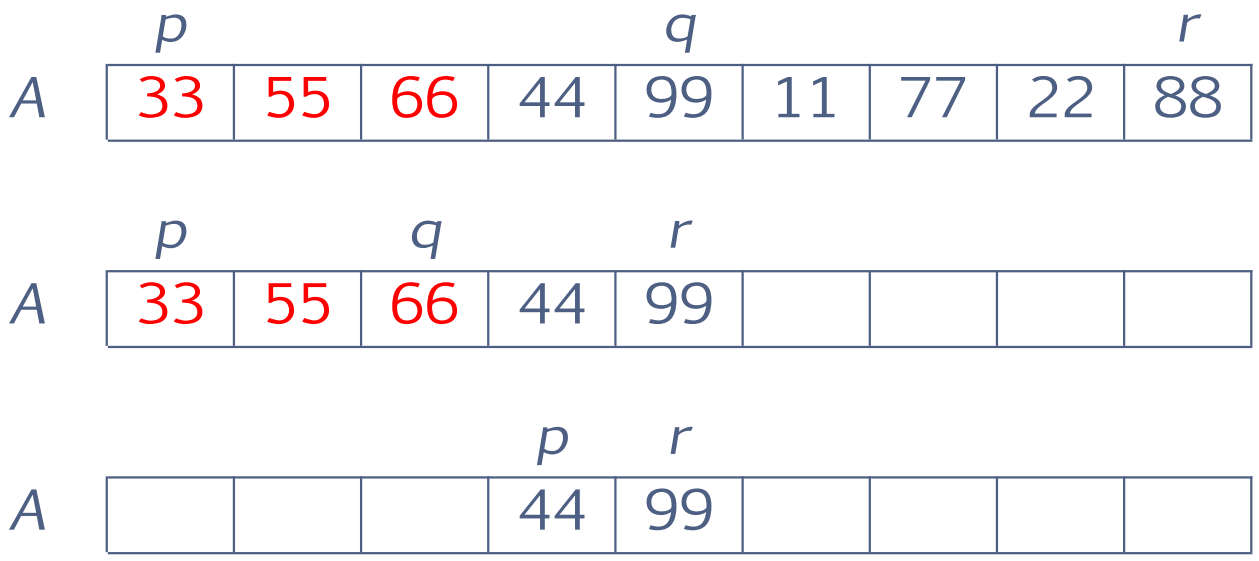
Ordenando por intercalación



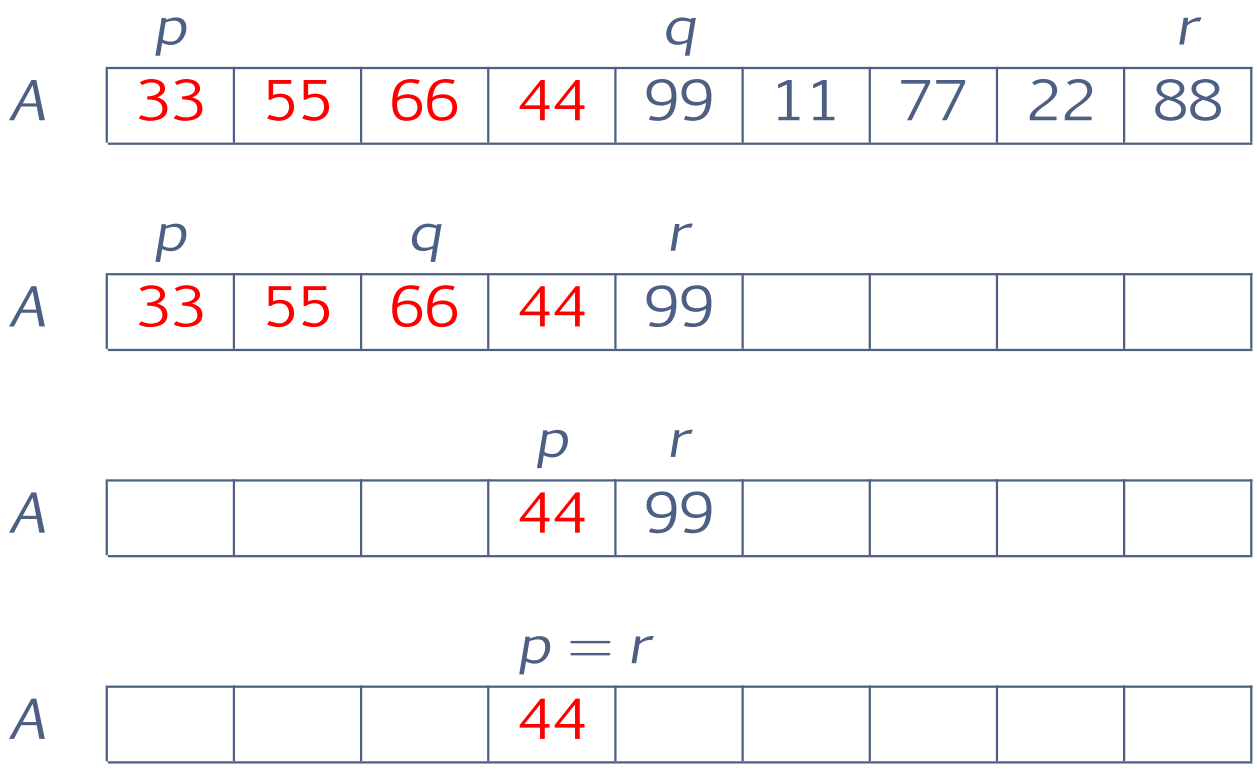
Ordenando por intercalación



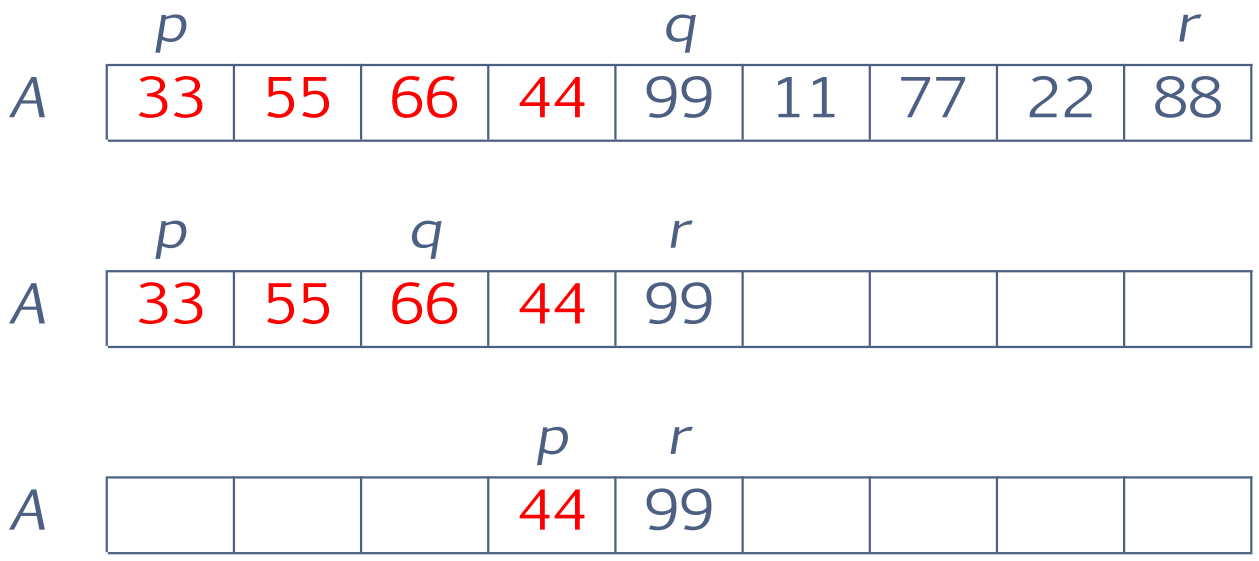
Ordenando por intercalación



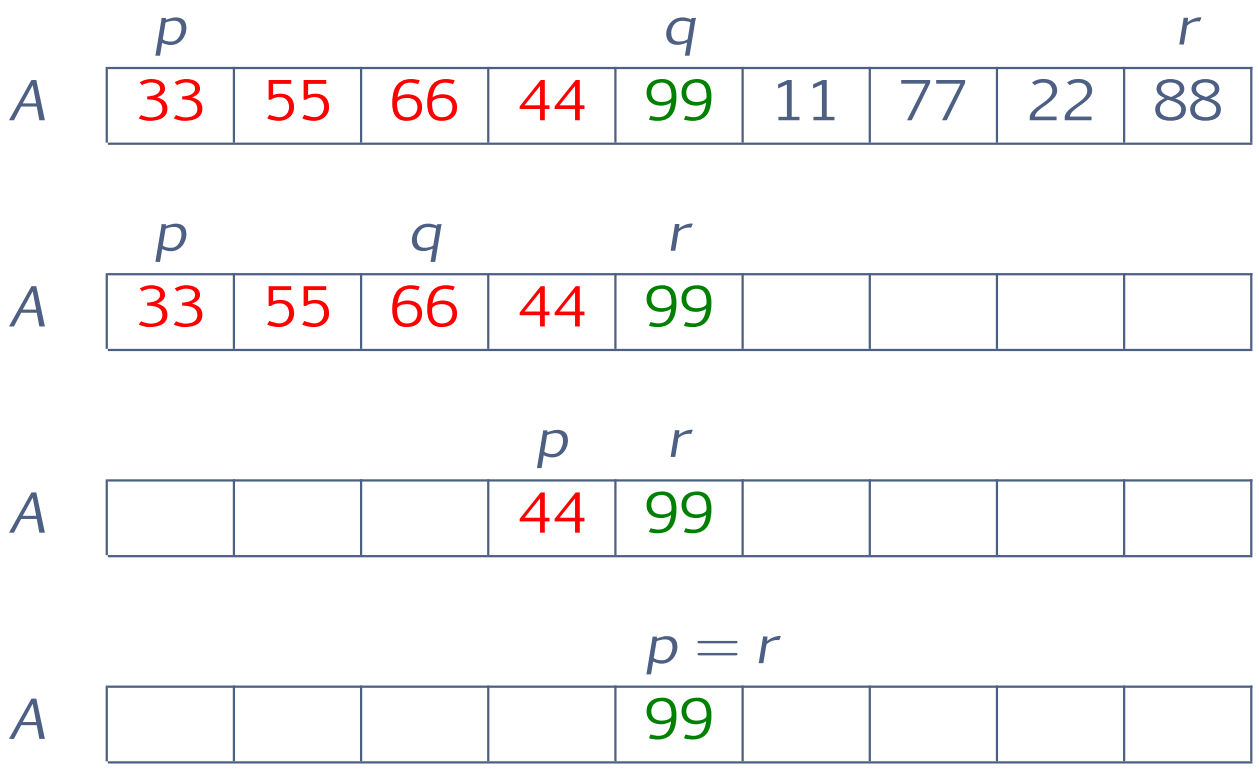
Ordenando por intercalación



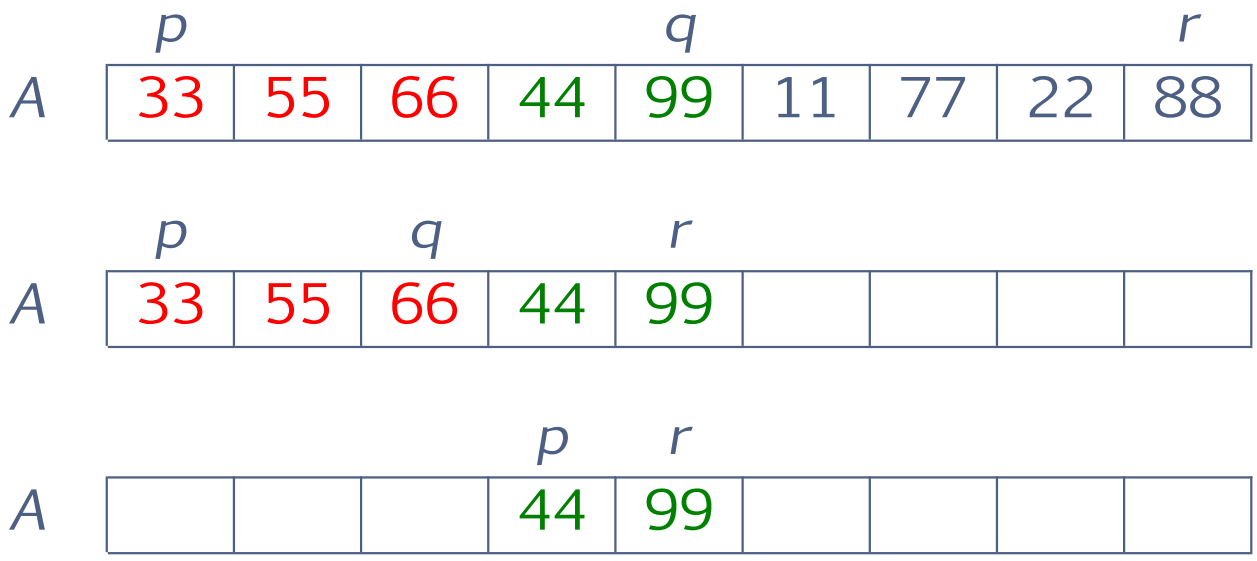
Ordenando por intercalación



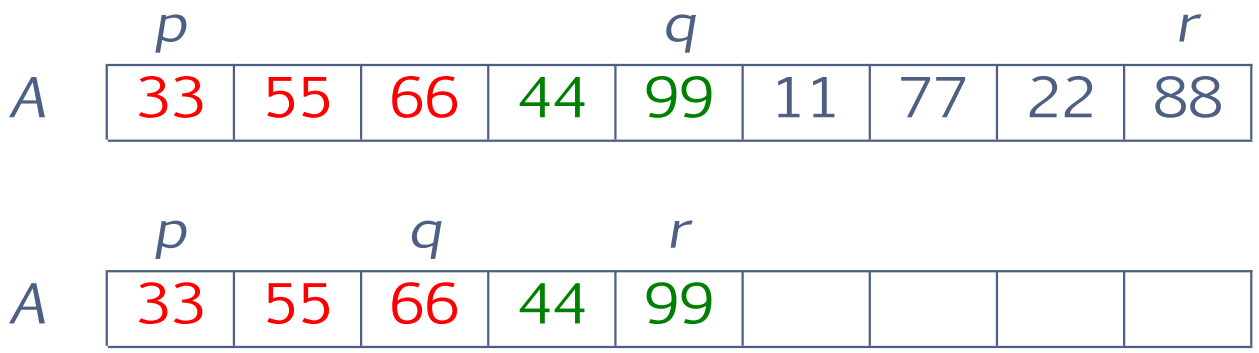
Ordenando por intercalación



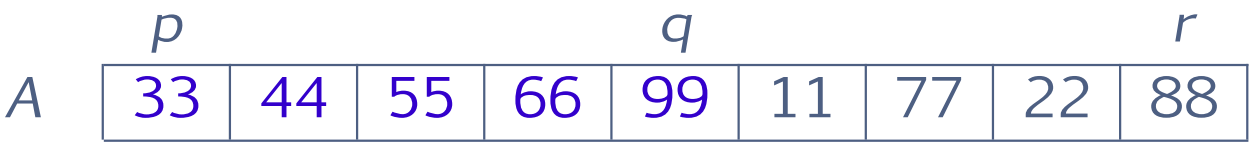
Ordenando por intercalación

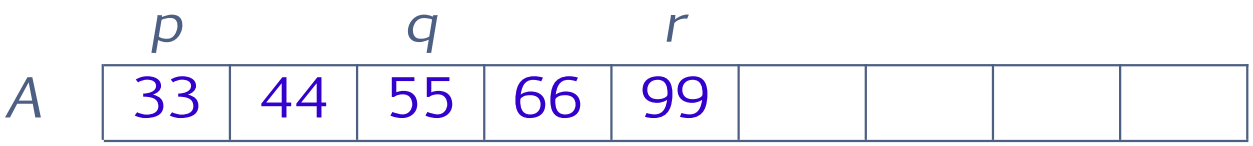


Ordenando por intercalación



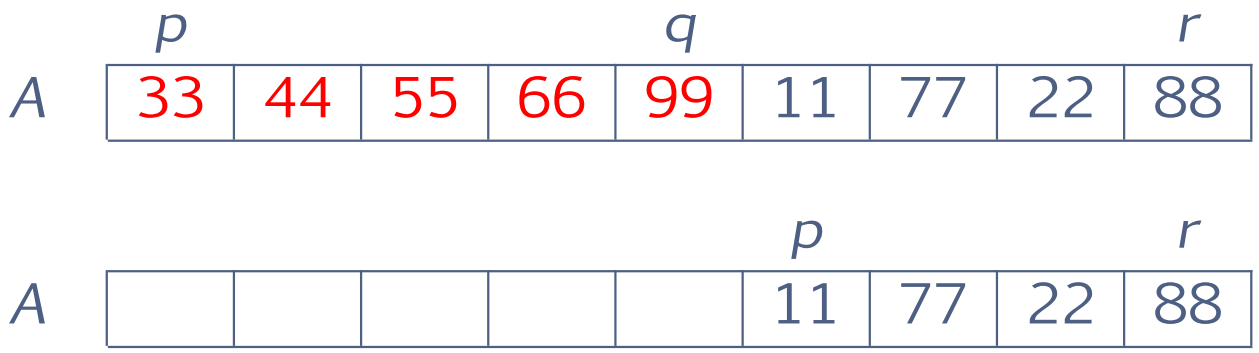
Ordenando por intercalación



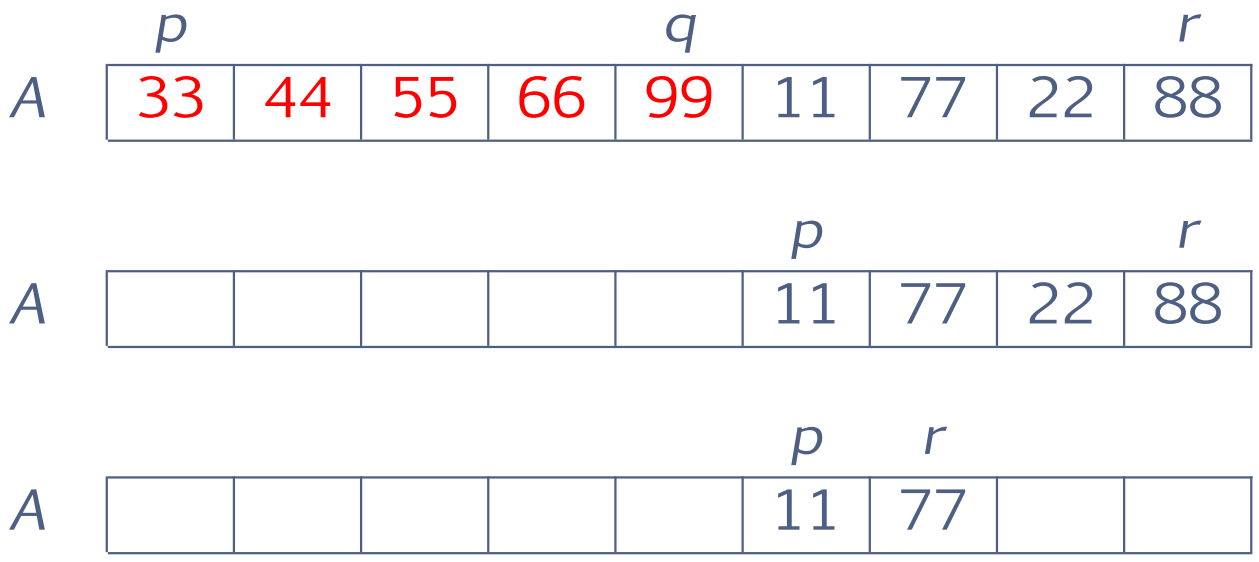


Ordenando por intercalación

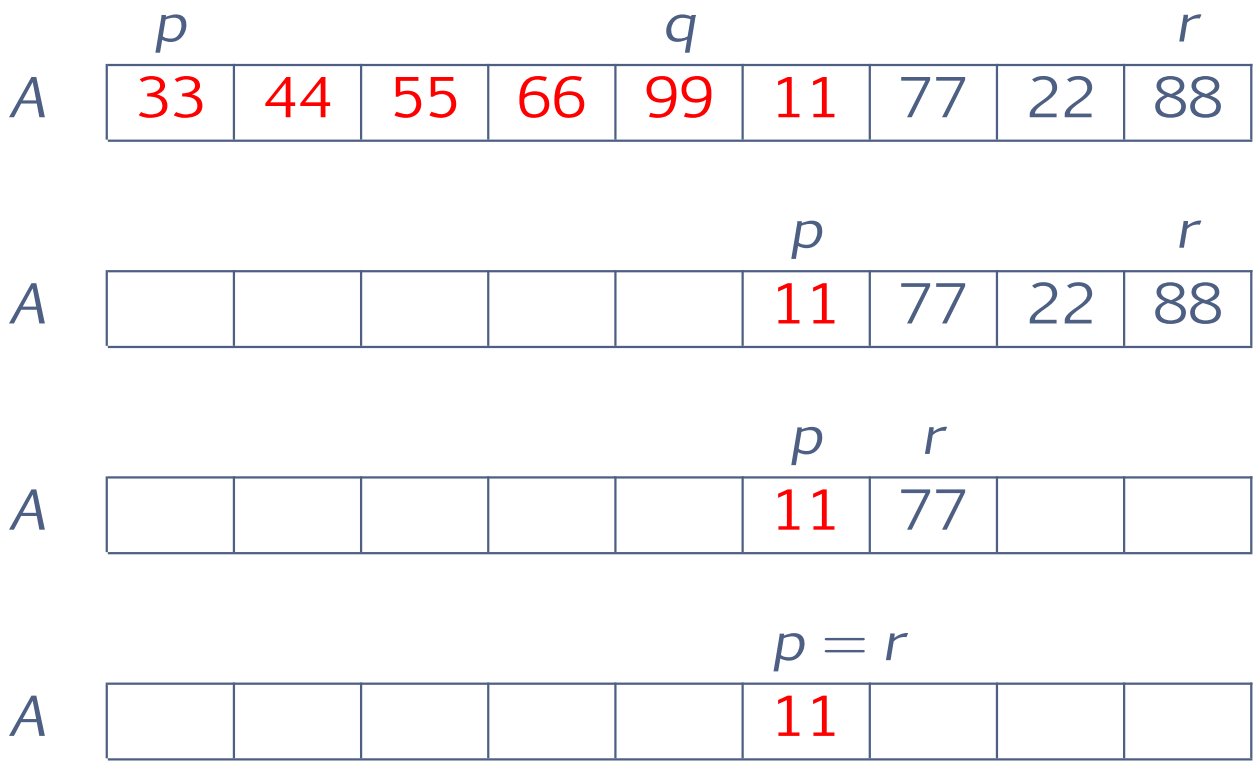
Ordenando por intercalación



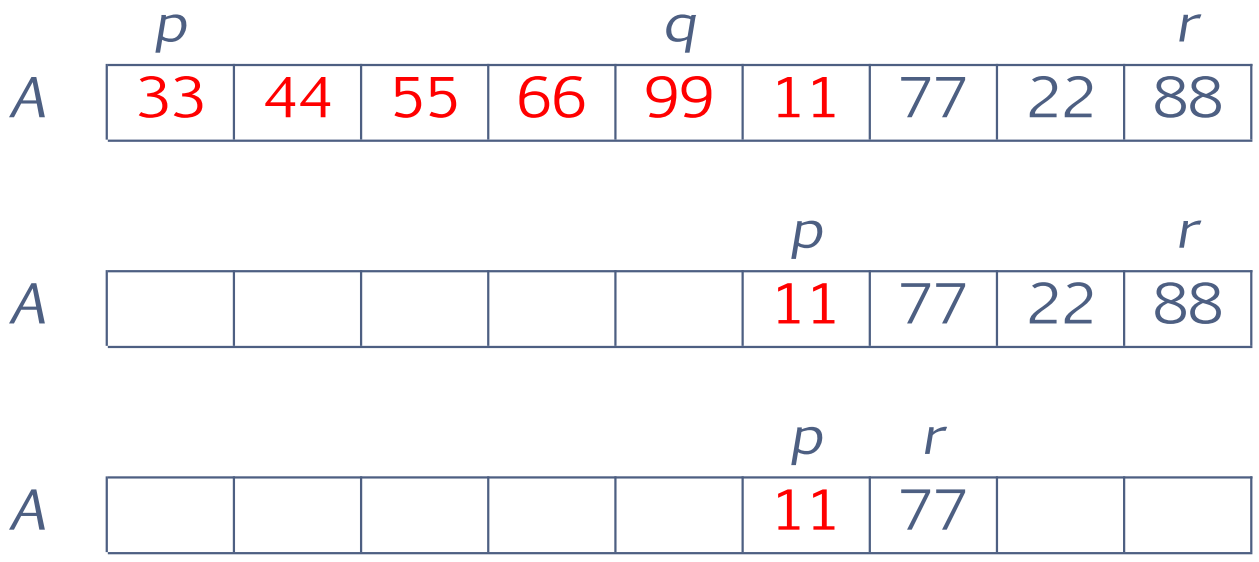
Ordenando por intercalación



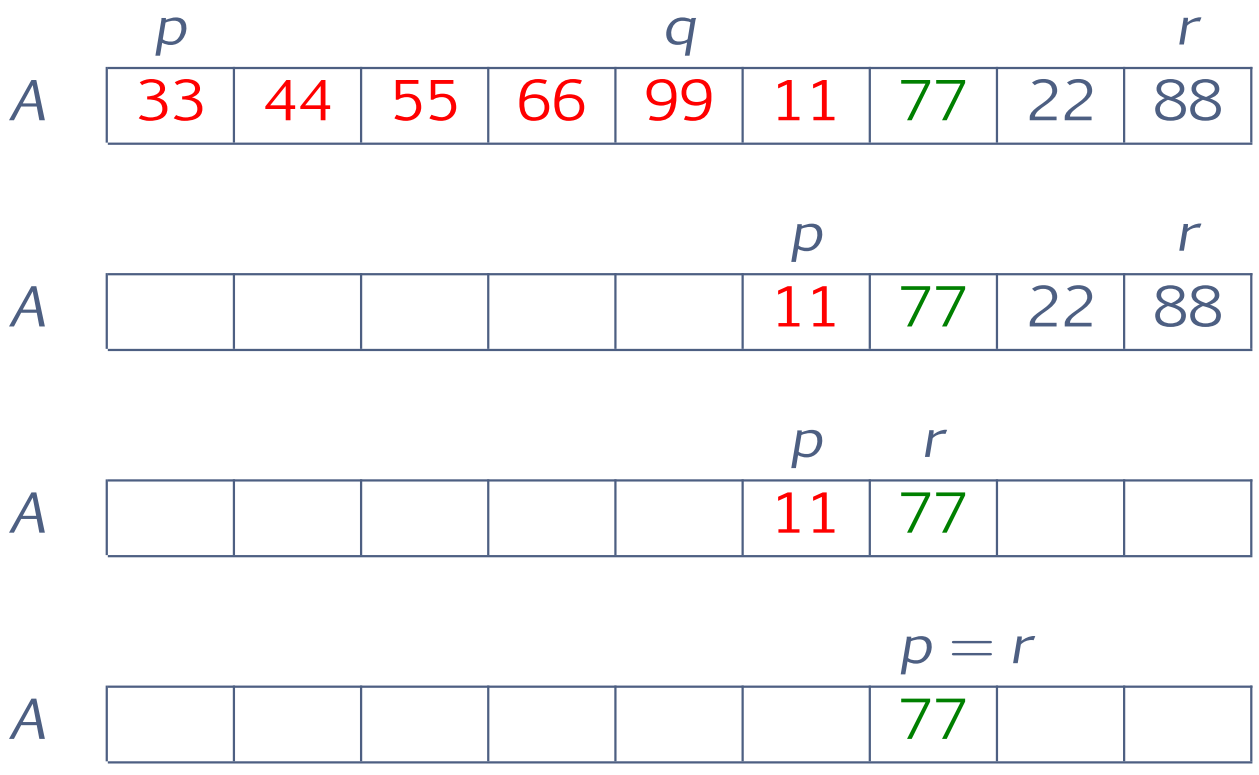
Ordenando por intercalación



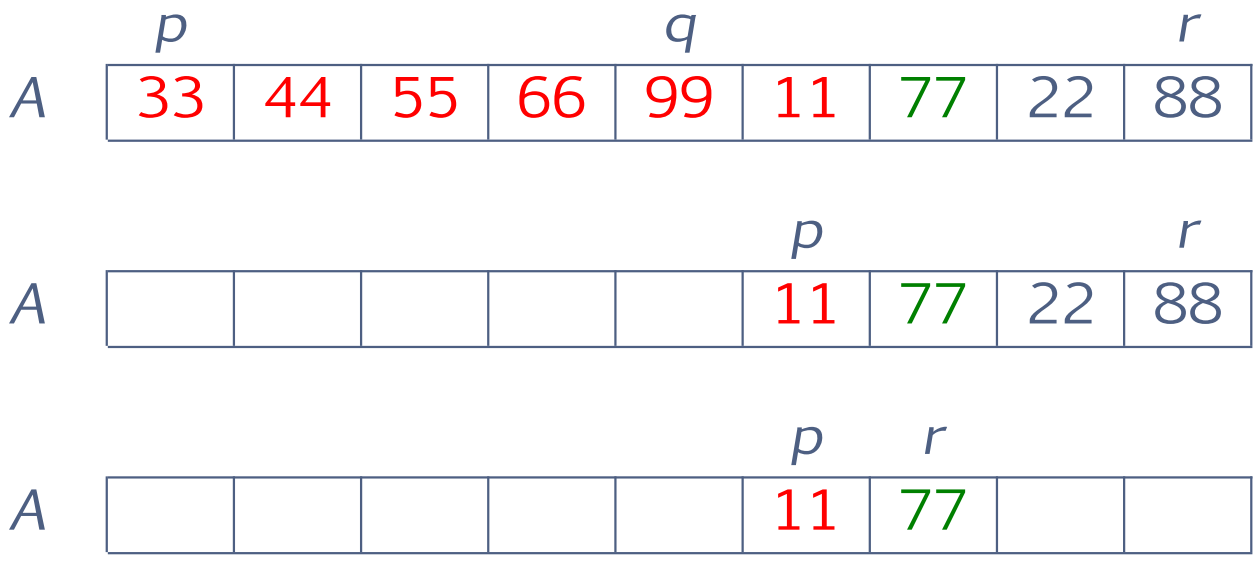
Ordenando por intercalación



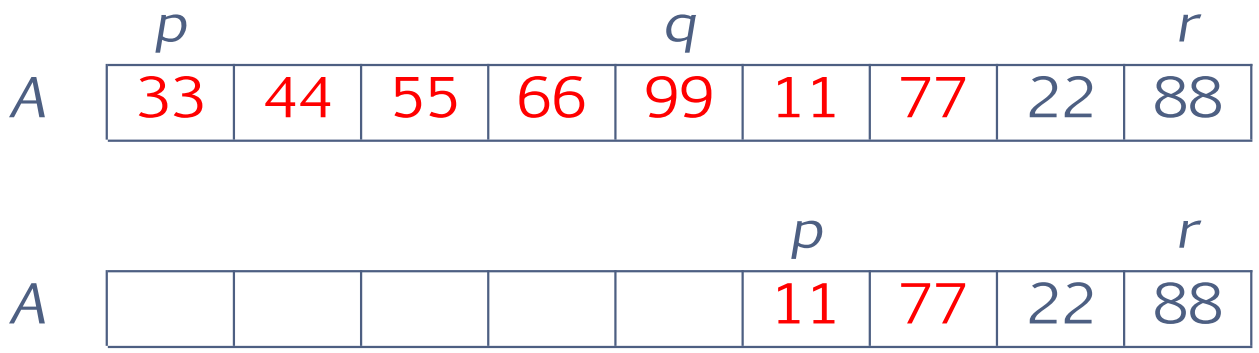
Ordenando por intercalación



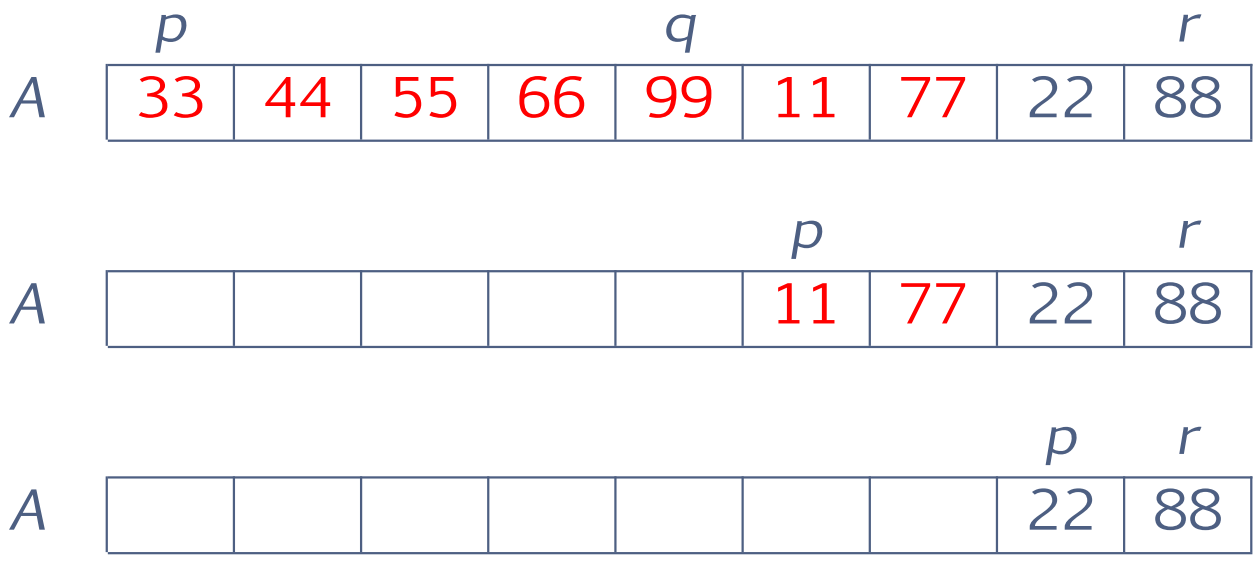
Ordenando por intercalación



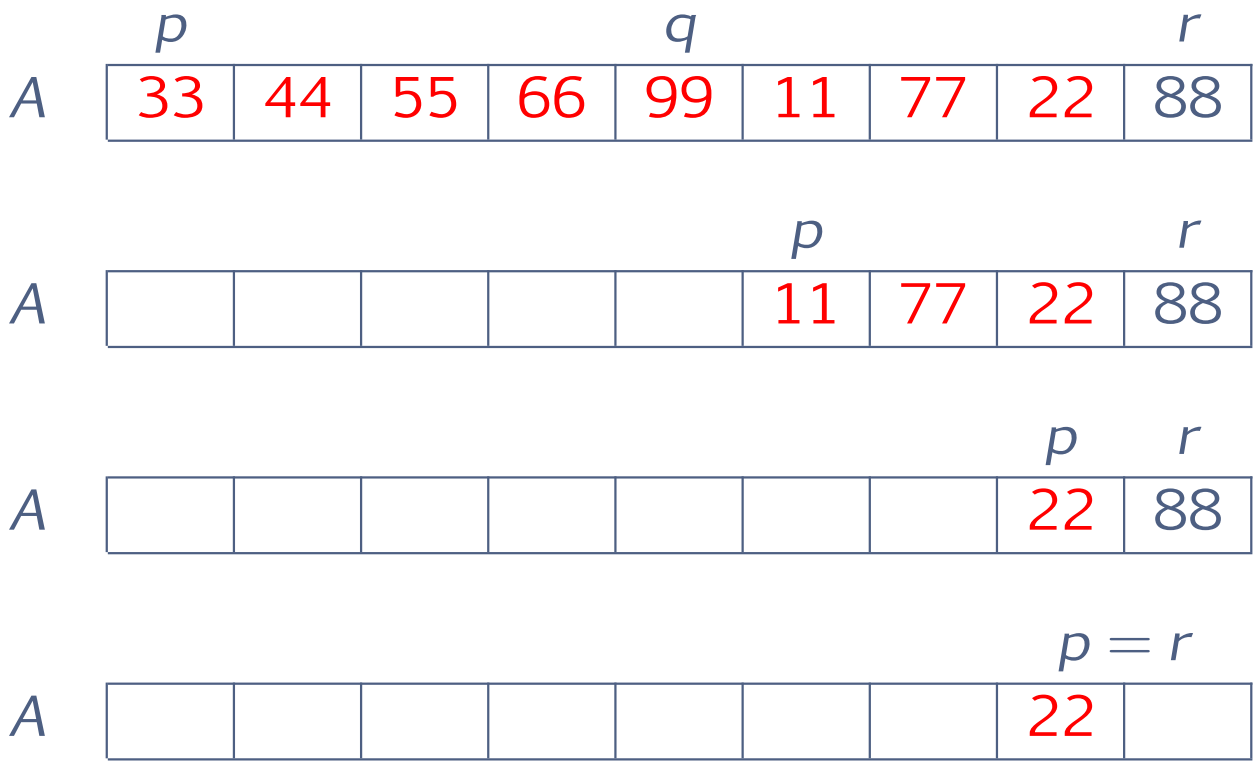
Ordenando por intercalación



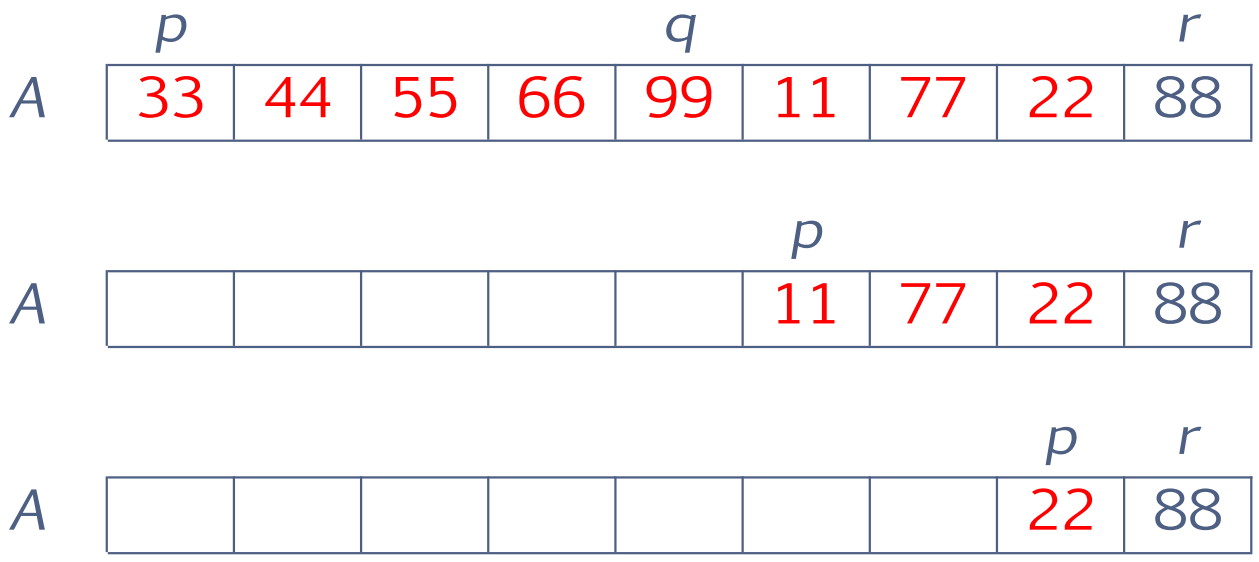
Ordenando por intercalación



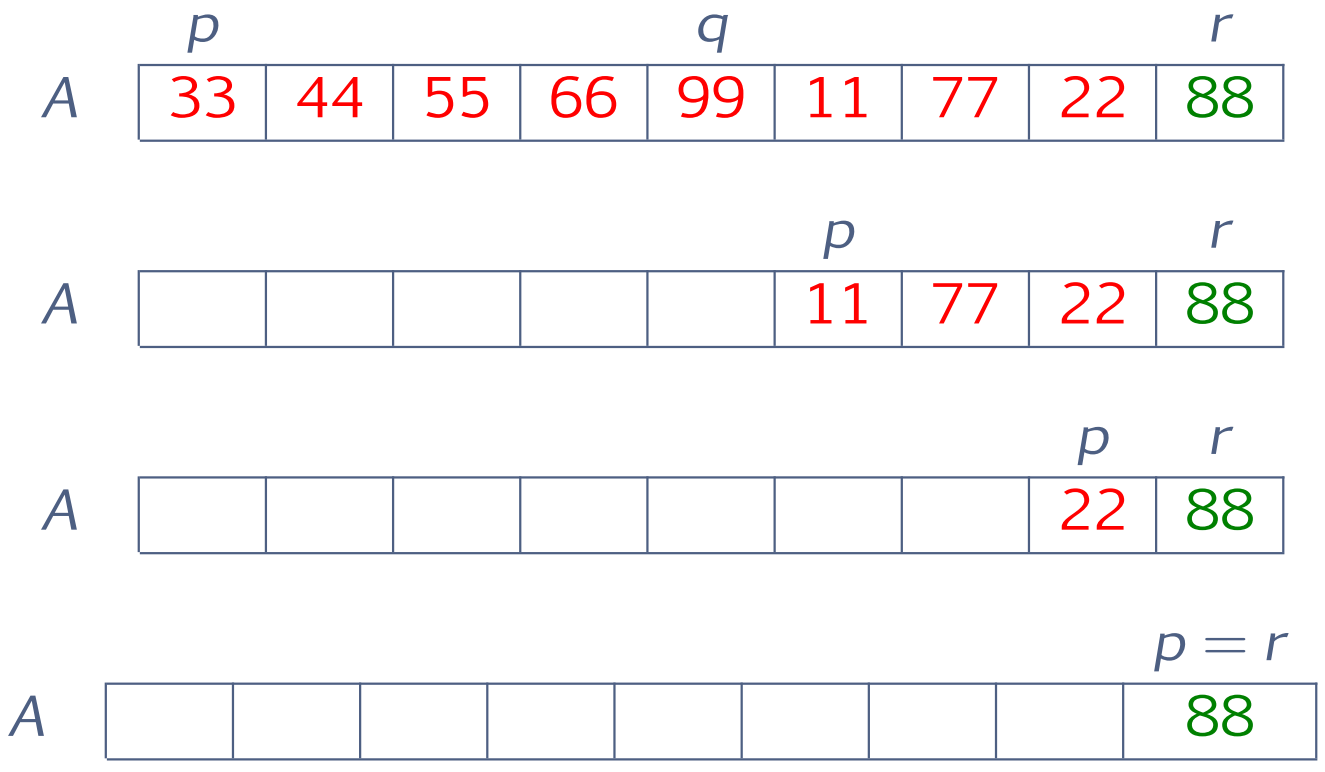
Ordenando por intercalación



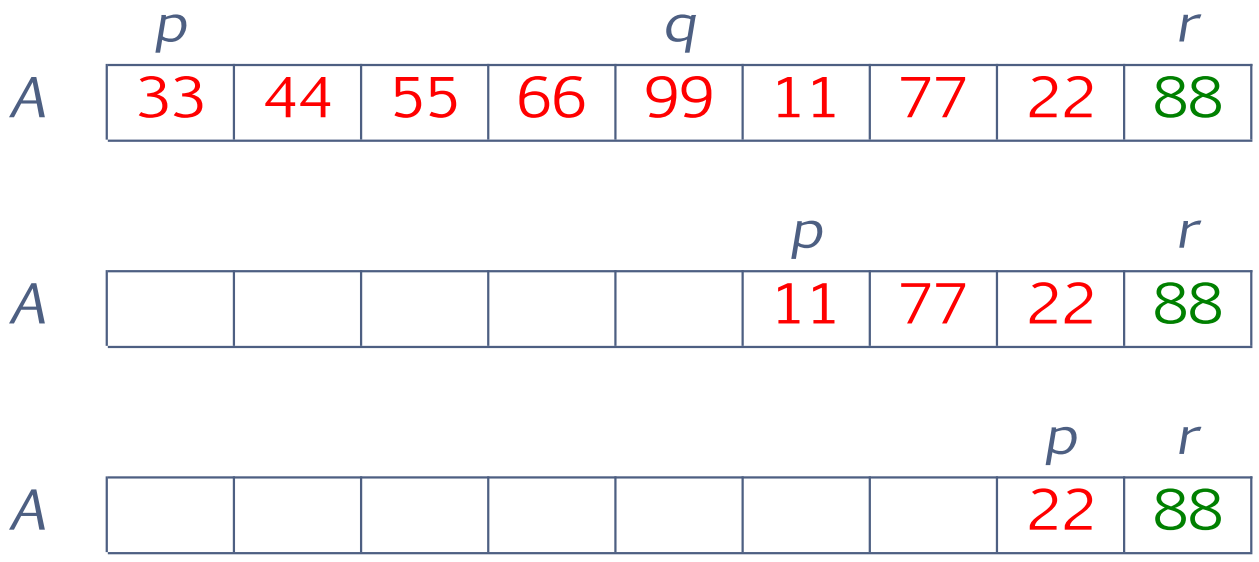
Ordenando por intercalación



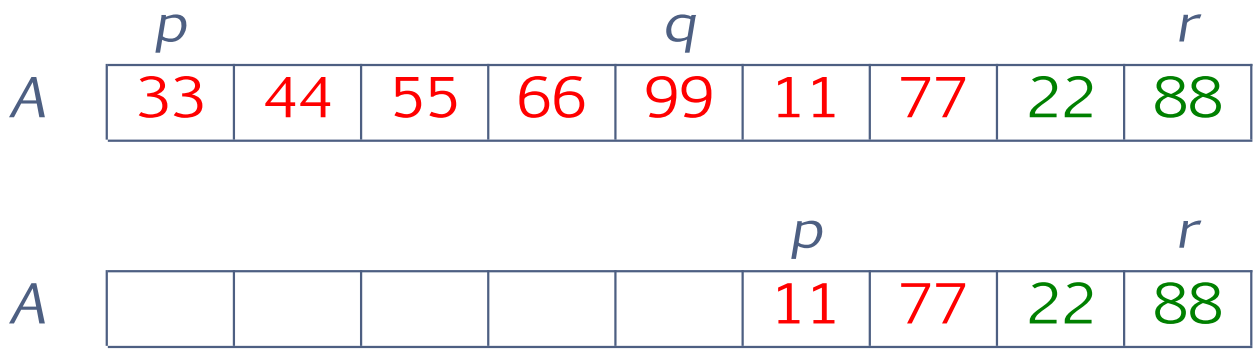
Ordenando por intercalación



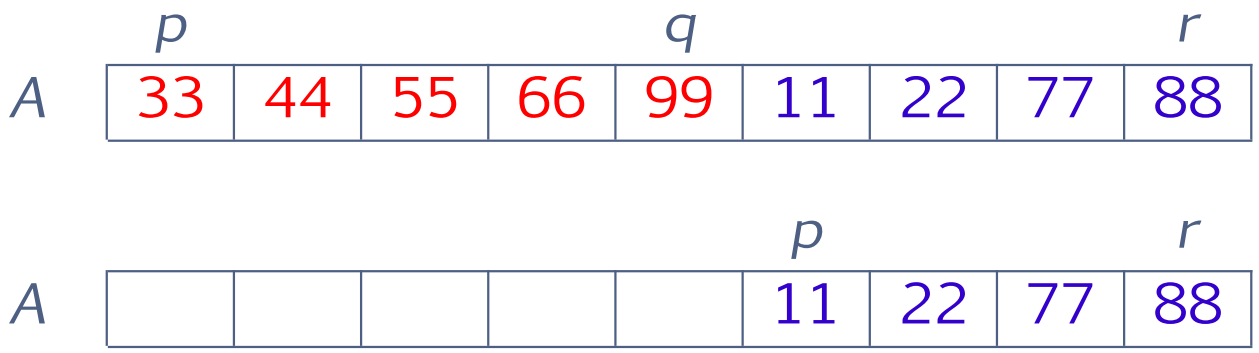
Ordenando por intercalación



Ordenando por intercalación



Ordenando por intercalación



Ordenando por intercalación

Ordenando por intercalación

Ordenando por intercalación

	p				q				r
A	11	22	33	44	55	66	77	88	99

Complejidad de Merge-Sort

- Tamaño de la entrada: $n = r - p + 1$
- Sea $T(n)$ el número de instrucciones ejecutadas en el peor caso

Complejidad de Merge-Sort

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

- Tamaño de la entrada: $n = r - p + 1$
- Sea $T(n)$ el número de instrucciones ejecutadas en el peor caso

Complejidad de Merge-Sort

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

- Tamaño de la entrada: $n = r - p + 1$
- Sea $T(n)$ el número de instrucciones ejecutadas en el peor caso

Complejidad de Merge-Sort

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

- Tamaño de la entrada: $n = r - p + 1$
- Sea $T(n)$ el número de instrucciones ejecutadas en el peor caso

Complejidad de MERGE-SORT

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

Línea	Tiempo
1	?
2	?
3	?
4	?
5	?

$$T(n) = ?$$

Complejidad de MERGE-SORT

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

Línea	Tiempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	$\Theta(n)$

$$T(n) = ?$$

Complejidad de Merge-Sort

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

Línea	Tiempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	$\Theta(n)$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) + \Theta(2)$$

Recurrencia

El tiempo de MERGE-SORT está dado por la fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtenemos una fórmula de recurrencia

- es la descripción de una función en términos de sí misma.
- el tiempo de un algoritmo recursivo acostumbra ser descrito por una recurrencia

Pero queremos una fórmula cerrada

- En ese caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver después las recurrencias

Recurrencia

El tiempo de MERGE-SORT está dado por la fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtenemos una fórmula de recurrencia

- es la descripción de una función en términos de sí misma.
- el tiempo de un algoritmo recursivo acostumbra ser descrito por una recurrencia

Pero queremos una fórmula cerrada

- En ese caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver después las recurrencias

Recurrencia

El tiempo de MERGE-SORT está dado por la fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtenemos una fórmula de recurrencia

- es la descripción de una función en términos de sí misma.
- el tiempo de un algoritmo recursivo acostumbra ser descrito por una recurrencia

Pero queremos una fórmula cerrada

- En ese caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver después las recurrencias

Recurrencia

El tiempo de MERGE-SORT está dado por la fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtenemos una fórmula de recurrencia

- es la descripción de una función en términos de sí misma.
- el tiempo de un algoritmo recursivo acostumbra ser descrito por una recurrencia

Pero queremos una fórmula cerrada

- En ese caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver después las recurrencias

Recurrencia

El tiempo de MERGE-SORT está dado por la fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtenemos una fórmula de recurrencia

- es la descripción de una función en términos de sí misma.
- el tiempo de un algoritmo recursivo acostumbra ser descrito por una recurrencia

Pero queremos una fórmula cerrada

- En ese caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver después las recurrencias

Recurrencia

El tiempo de MERGE-SORT está dado por la fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtenemos una fórmula de recurrencia

- es la descripción de una función en términos de sí misma.
- el tiempo de un algoritmo recursivo acostumbra ser descrito por una recurrencia

Pero queremos una fórmula cerrada

- En ese caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver después las recurrencias

Recurrencia

El tiempo de MERGE-SORT está dado por la fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtenemos una fórmula de recurrencia

- es la descripción de una función en términos de sí misma.
- el tiempo de un algoritmo recursivo acostumbra ser descrito por una recurrencia

Pero queremos una fórmula cerrada

- En ese caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver después las recurrencias

Recurrencia

El tiempo de MERGE-SORT está dado por la fórmula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n = 2, 3, 4, \ldots \end{cases}$$

Obtenemos una fórmula de recurrencia

- es la descripción de una función en términos de sí misma.
- el tiempo de un algoritmo recursivo acostumbra ser descrito por una recurrencia

Pero queremos una fórmula cerrada

- En ese caso, $T(n) = \Theta(n \log n)$
- Aprenderemos a resolver después las recurrencias

Proyectando Algoritmos

- Entendiendo la importancia de algoritmos eficientes

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1\,GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10\,MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1\,GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10\,MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1\,GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10\,MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos

Que diferencia hace tener un algoritmo $\Theta(n^2)$ o $\Theta(n \log n)$?

Suponga que queremos ordenar un vector de n elementos

1. Usando una computadora A con velocidad de $1GHz$ y con un algoritmo que ejecuta $2n^2$ instrucciones.
2. Usando una computadora B con velocidad de $10MHz$ y con un algoritmo que ejecuta $50n \log n$ instrucciones.

Comparando las implementaciones:

- La computadora A es 100 veces mas rápida que la computadora B.
- La constante multiplicativa de la segunda es mucho mayor:
  - puede ser debido a un lenguaje de mas alto nível
  - o debido a un programador con menos experiencia

Algoritmos buenos y malos (cont)

Que implementación ordena un millón de elementos primero?

1. La máquina A demora

$$\frac{2 \cdot (10^6)^2 \text{instrucciones}}{10^9 \text{instrucciones /segundo}} \approx 2000 \text{segundos}$$

2. La maquina B demora

$$\frac{50 \cdot (10^6 \log 10^6) \text{instrucciones}}{10^7 \text{instrucciones /segundo}} \approx 100 \text{segundos}$$

- La máquina B fue veinte veces mas rápida que la máquina A
- Si fuesen 10 milloness de elementos, la proporción sería de 2,3 días contra 20 minutos

Algoritmos buenos y malos (cont)

Que implementación ordena un millón de elementos primero?

1. La máquina A demora

$$\frac{2 \cdot (10^6)^2 \text{instrucciones}}{10^9 \text{instrucciones /segundo}} \approx 2000 \text{segundos}$$

2. La maquina B demora

$$\frac{50 \cdot (10^6 \log 10^6) \text{instrucciones}}{10^7 \text{instrucciones /segundo}} \approx 100 \text{segundos}$$

- La máquina B fue veinte veces mas rápida que la máquina A
- Si fuesen 10 milloness de elementos, la proporción sería de 2,3 días contra 20 minutos

Algoritmos buenos y malos (cont)

Que implementación ordena un millón de elementos primero?

1. La máquina A demora

$$\frac{2 \cdot (10^6)^2 \text{instrucciones}}{10^9 \text{instrucciones /segundo}} \approx 2000 \text{segundos}$$

2. La maquina B demora

$$\frac{50 \cdot (10^6 \log 10^6) \text{instrucciones}}{10^7 \text{instrucciones /segundo}} \approx 100 \text{segundos}$$

- La máquina B fue veinte veces mas rápida que la máquina A
- Si fuesen 10 milloness de elementos, la proporción sería de 2,3 días contra 20 minutos

Algoritmos buenos y malos (cont)

Que implementación ordena un millón de elementos primero?

1. La máquina A demora

$$\frac{2 \cdot (10^6)^2 \text{instrucciones}}{10^9 \text{instrucciones /segundo}} \approx 2000 \text{segundos}$$

2. La maquina B demora

$$\frac{50 \cdot (10^6 \log 10^6) \text{instrucciones}}{10^7 \text{instrucciones /segundo}} \approx 100 \text{segundos}$$

- La máquina B fue veinte veces mas rápida que la máquina A
- Si fuesen 10 milloness de elementos, la proporción sería de 2,3 días contra 20 minutos

Algoritmos buenos y malos (cont)

Que implementación ordena un millón de elementos primero?

1. La máquina A demora

$$\frac{2 \cdot (10^6)^2 \text{instrucciones}}{10^9 \text{instrucciones /segundo}} \approx 2000 \text{segundos}$$

2. La maquina B demora

$$\frac{50 \cdot (10^6 \log 10^6) \text{instrucciones}}{10^7 \text{instrucciones /segundo}} \approx 100 \text{segundos}$$

- La máquina B fue veinte veces mas rápida que la máquina A
- Si fuesen 10 milloness de elementos, la proporción sería de 2,3 días contra 20 minutos

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Algoritmos y tecnología

Y ¿si usaramos una supercomputadora?

$f(n)$	Computador atual	100×mas rápido	1000×mas rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Si desarrollamos un algoritmo malo, no podremos resolver problemas complejos.

Conclusiones

Algunas conclusiones:

- Hacer proyectos mejores algoritmos puede llevar a ganancias extraordinárias de desempeño.
- Un buen proyecto de algoritmos es tan importante cuanto el proyecto de hardware.
- La ganancia obtenida al mejorar la complejidad de un algoritmo no podría ser obtenida simplemente con el avance de la tecnología.
- Queremos estudiar principalmente algoritmos fundamentales, que producen avances en otros componentes básicos de las aplicaciones (piense en los compiladores, buscadores en la internet, clasificadores, etc).

Conclusiones

Algunas conclusiones:

- Hacer proyectos mejores algoritmos puede llevar a ganancias extraordinárias de desempeño.
- Un buen proyecto de algoritmos es tan importante cuanto el proyecto de hardware.
- La ganancia obtenida al mejorar la complejidad de un algoritmo no podría ser obtenida simplemente con el avance de la tecnología.
- Queremos estudiar principalmente algoritmos fundamentales, que producen avances en otros componentes básicos de las aplicaciones (piense en los compiladores, buscadores en la internet, clasificadores, etc).

Conclusiones

Algunas conclusiones:

- Hacer proyectos mejores algoritmos puede llevar a ganancias extraordinárias de desempeño.
- Un buen proyecto de algoritmos es tan importante cuanto el proyecto de hardware.
- La ganancia obtenida al mejorar la complejidad de un algoritmo no podría ser obtenida simplemente con el avance de la tecnología.
- Queremos estudiar principalmente algoritmos fundamentales, que producen avances en otros componentes básicos de las aplicaciones (piense en los compiladores, buscadores en la internet, clasificadores, etc).

Conclusiones

Algunas conclusiones:

- Hacer proyectos mejores algoritmos puede llevar a ganancias extraordinárias de desempeño.
- Un buen proyecto de algoritmos es tan importante cuanto el proyecto de hardware.
- La ganancia obtenida al mejorar la complejidad de un algoritmo no podría ser obtenida simplemente con el avance de la tecnología.
- Queremos estudiar principalmente algoritmos fundamentales, que producen avances en otros componentes básicos de las aplicaciones (piense en los compiladores, buscadores en la internet, clasificadores, etc).

Conclusiones

Algunas conclusiones:

- Hacer proyectos mejores algoritmos puede llevar a ganancias extraordinárias de desempeño.
- Un buen proyecto de algoritmos es tan importante cuanto el proyecto de hardware.
- La ganancia obtenida al mejorar la complejidad de un algoritmo no podría ser obtenida simplemente con el avance de la tecnología.
- Queremos estudiar principalmente algoritmos fundamentales, que producen avances en otros componentes básicos de las aplicaciones (piense en los compiladores, buscadores en la internet, clasificadores, etc).