Programación III

Ricardo Wehbe

UADE

30 de agosto de 2021

Programa

Welcome back my friends to the show that never ends

Emerson, Lake & Palmer, *Karn Evil 9*

El juego de las reglas

El juego de las reglas

- Hay dos exámenes parciales. Los parciales tendrán lugar los días 4 de octubre y 15 de noviembre. Los estudiantes pueden ser llamados para ser interrogados sobre algunos puntos de los parciales.

El juego de las reglas

- Hay dos exámenes parciales. Los parciales tendrán lugar los días 4 de octubre y 15 de noviembre. Los estudiantes pueden ser llamados para ser interrogados sobre algunos puntos de los parciales.
- Hay ejercicios sugeridos. Los estudiantes serán invitados aleatoriamente a exponer sus soluciones.

El juego de las reglas

- Hay dos exámenes parciales. Los parciales tendrán lugar los días 4 de octubre y 15 de noviembre. Los estudiantes pueden ser llamados para ser interrogados sobre algunos puntos de los parciales.
- Hay ejercicios sugeridos. Los estudiantes serán invitados aleatoriamente a exponer sus soluciones.
- Hay un examen recuperatorio de uno de los parciales. El recuperatorio tendrá lugar el día 29 de noviembre.

El juego de las reglas

- Hay dos exámenes parciales. Los parciales tendrán lugar los días 4 de octubre y 15 de noviembre. Los estudiantes pueden ser llamados para ser interrogados sobre algunos puntos de los parciales.
- Hay ejercicios sugeridos. Los estudiantes serán invitados aleatoriamente a exponer sus soluciones.
- Hay un examen recuperatorio de uno de los parciales. El recuperatorio tendrá lugar el día 29 de noviembre.
- Existe la posibilidad de rendir el examen final anticipadamente el día 29 de noviembre. Esta posibilidad está reservada para los que hubieren aprobado ambos parciales.

El juego de las reglas

- Hay dos exámenes parciales. Los parciales tendrán lugar los días 4 de octubre y 15 de noviembre. Los estudiantes pueden ser llamados para ser interrogados sobre algunos puntos de los parciales.
- Hay ejercicios sugeridos. Los estudiantes serán invitados aleatoriamente a exponer sus soluciones.
- Hay un examen recuperatorio de uno de los parciales. El recuperatorio tendrá lugar el día 29 de noviembre.
- Existe la posibilidad de rendir el examen final anticipadamente el día 29 de noviembre. Esta posibilidad está reservada para los que hubieren aprobado ambos parciales.
- El examen final regular tendrá lugar el día 13 de diciembre.

Las reglas del juego

Las reglas del juego

- Para aprobar la cursada es necesario aprobar ambos parciales o bien uno de los parciales y el recuperatorio del otro. Observe que hay una única instancia de recuperación.

Las reglas del juego

- Para aprobar la cursada es necesario aprobar ambos parciales o bien uno de los parciales y el recuperatorio del otro. Observe que hay una única instancia de recuperación.
- Para aprobar el examen final, usted debe demostrar que maneja las técnicas que se estudiarán en el curso.

Las reglas del juego

- Para aprobar la cursada es necesario aprobar ambos parciales o bien uno de los parciales y el recuperatorio del otro. Observe que hay una única instancia de recuperación.
- Para aprobar el examen final, usted debe demostrar que maneja las técnicas que se estudiarán en el curso.
- Todas las preguntas sobre temas del curso o ejercicios se concentrarán en los foros. Hay varios foros separados por tema.

Las reglas del juego

- Para aprobar la cursada es necesario aprobar ambos parciales o bien uno de los parciales y el recuperatorio del otro. Observe que hay una única instancia de recuperación.
- Para aprobar el examen final, usted debe demostrar que maneja las técnicas que se estudiarán en el curso.
- Todas las preguntas sobre temas del curso o ejercicios se concentrarán en los foros. Hay varios foros separados por tema.
- Las comunicaciones a través de la mensajería de Teams se limitarán a cuestiones urgentes. Preguntas sobre ejercicios efectuadas por Teams serán respondidas en el foro correspondiente.

Las reglas del juego

- Para aprobar la cursada es necesario aprobar ambos parciales o bien uno de los parciales y el recuperatorio del otro. Observe que hay una única instancia de recuperación.
- Para aprobar el examen final, usted debe demostrar que maneja las técnicas que se estudiarán en el curso.
- Todas las preguntas sobre temas del curso o ejercicios se concentrarán en los foros. Hay varios foros separados por tema.
- Las comunicaciones a través de la mensajería de Teams se limitarán a cuestiones urgentes. Preguntas sobre ejercicios efectuadas por Teams serán respondidas en el foro correspondiente.
- Las preguntas sobre ejercicios deben ser complementadas con una "proof of work" que muestre que hubo un intento de resolución del ejercicio. Presguntas que sonsistan de 200 líneas de código acompañadas de la frase "me da error" son inaceptables.

Algunas malas noticias

Algunas malas noticias

- Ésta es una materia difícil.

Algunas malas noticias

- Ésta es una materia difícil.
- ¿Se puede aprender a nadar en un aula oyendo hablar a un instructor?

Algunas malas noticias

- Ésta es una materia difícil.
- ¿Se puede aprender a nadar en un aula oyendo hablar a un instructor?
- Tampoco a programar.

Algunas malas noticias

- Ésta es una materia difícil.
- ¿Se puede aprender a nadar en un aula oyendo hablar a un instructor?
- Tampoco a programar.
- Algunos problemas observados:
    - Lógica. Hay una cierta deficiencia de pensamiento lógico.
    - Análisis. Se tiende a resolver problemas mecánicamente sin buscar llegar a una comprensión del problema primero.
    - Síntesis. No hay un buen relacionamiento entre los temas.

Algunas malas noticias

- Ésta es una materia difícil.
- ¿Se puede aprender a nadar en un aula oyendo hablar a un instructor?
- Tampoco a programar.
- Algunos problemas observados:
  - Lógica. Hay una cierta deficiencia de pensamiento lógico.
  - Análisis. Se tiende a resolver problemas mecánicamente sin buscar llegar a una comprensión del problema primero.
  - Síntesis. No hay un buen relacionamiento entre los temas.
- ¿Y por qué me cuentan todo esto?

Algunas malas noticias

- Ésta es una materia difícil.
- ¿Se puede aprender a nadar en un aula oyendo hablar a un instructor?
- Tampoco a programar.
- Algunos problemas observados:
  - Lógica. Hay una cierta deficiencia de pensamiento lógico.
  - Análisis. Se tiende a resolver problemas mecánicamente sin buscar llegar a una comprensión del problema primero.
  - Síntesis. No hay un buen relacionamiento entre los temas.
- ¿Y por qué me cuentan todo esto?
- Porque justamente esto es lo que se evalúa.

¿De qué trata esta materia?

¿De qué trata esta materia?

- Estudio de diferentes técnicas de construcción de algoritmos para encontrar la más eficiente desde el punto de vista de la complejidad temporal para un problema dado.

¿De qué trata esta materia?

- Estudio de diferentes técnicas de construcción de algoritmos para encontrar la más eficiente desde el punto de vista de la complejidad temporal para un problema dado.
- Análisis de eficiencia temporal.

¿De qué trata esta materia?

- Estudio de diferentes técnicas de construcción de algoritmos para encontrar la más eficiente desde el punto de vista de la complejidad temporal para un problema dado.
- Análisis de eficiencia temporal.

Nuestro objeto de estudio son entonces los *algoritmos*.

¿De qué trata esta materia?

- Estudio de diferentes técnicas de construcción de algoritmos para encontrar la más eficiente desde el punto de vista de la complejidad temporal para un problema dado.
- Análisis de eficiencia temporal.

Nuestro objeto de estudio son entonces los *algoritmos*.
Las técnicas que estudiaremos son (por orden de aparición):

¿De qué trata esta materia?

- Estudio de diferentes técnicas de construcción de algoritmos para encontrar la más eficiente desde el punto de vista de la complejidad temporal para un problema dado.
- Análisis de eficiencia temporal.

Nuestro objeto de estudio son entonces los *algoritmos*.
Las técnicas que estudiaremos son (por orden de aparición):

- *Divide and conquer*

¿De qué trata esta materia?

- Estudio de diferentes técnicas de construcción de algoritmos para encontrar la más eficiente desde el punto de vista de la complejidad temporal para un problema dado.
- Análisis de eficiencia temporal.

Nuestro objeto de estudio son entonces los *algoritmos*.
Las técnicas que estudiaremos son (por orden de aparición):

- *Divide and conquer*
- Algoritmos "voraces" (*Greedy algorithms*)

¿De qué trata esta materia?

- Estudio de diferentes técnicas de construcción de algoritmos para encontrar la más eficiente desde el punto de vista de la complejidad temporal para un problema dado.
- Análisis de eficiencia temporal.

Nuestro objeto de estudio son entonces los *algoritmos*.
Las técnicas que estudiaremos son (por orden de aparición):

- *Divide and conquer*
- Algoritmos "voraces" (*Greedy algorithms*)
- Programación dinámica

¿De qué trata esta materia?

- Estudio de diferentes técnicas de construcción de algoritmos para encontrar la más eficiente desde el punto de vista de la complejidad temporal para un problema dado.
- Análisis de eficiencia temporal.

Nuestro objeto de estudio son entonces los *algoritmos*.
Las técnicas que estudiaremos son (por orden de aparición):

- *Divide and conquer*
- Algoritmos "voraces" (*Greedy algorithms*)
- Programación dinámica
- *Backtracking*

¿Qué es un algoritmo?

¿Qué es un algoritmo?

- Primer intento de formalización: David Hilbert, a comienzos del siglo XX ("ein Verfahren", un procedimiento.)

¿Qué es un algoritmo?

- Primer intento de formalización: David Hilbert, a comienzos del siglo XX ("ein Verfahren", un procedimiento.)
- Un algoritmo es una secuencia ordenada y finita de acciones que transforman un conjunto de datos de entrada en datos de salida y que resuelven un problema específico.

¿Qué es un algoritmo?

- Primer intento de formalización: David Hilbert, a comienzos del siglo XX ("ein Verfahren", un procedimiento.)
- Un algoritmo es una secuencia ordenada y finita de acciones que transforman un conjunto de datos de entrada en datos de salida y que resuelven un problema específico.
- Un algoritmo tiene cero o más datos de entrada, al menos una salida, cada paso esta unívocamente determinado y debe terminar en un número finito de pasos.

¿Qué es un algoritmo?

- Primer intento de formalización: David Hilbert, a comienzos del siglo XX ("ein Verfahren", un procedimiento.)
- Un algoritmo es una secuencia ordenada y finita de acciones que transforman un conjunto de datos de entrada en datos de salida y que resuelven un problema específico.
- Un algoritmo tiene cero o más datos de entrada, al menos una salida, cada paso esta unívocamente determinado y debe terminar en un número finito de pasos.
- Estas definiciones son para esta materia; en general, algunos algoritmos pueden no terminar y en algunos casos cada paso puede *no* estar unívocamente determinado.

Corrección y eficiencia de algoritmos. Informalmente

Corrección y eficiencia de algoritmos. Informalmente

- Corrección: ¿resuelve el algoritmo nuestro problema?

Corrección y eficiencia de algoritmos. Informalmente

- Corrección: ¿resuelve el algoritmo nuestro problema?
- Eficiencia: ¿cuán rápidamente lo hace?

Corrección y eficiencia de algoritmos. Informalmente

- Corrección: ¿resuelve el algoritmo nuestro problema?
- Eficiencia: ¿cuán rápidamente lo hace?
- Si para contar vacas les contamos las patas y dividimos el resultado por cuatro, estamos resolviendo el problema. Pero no de la manera más eficiente.

Corrección y eficiencia de algoritmos. Informalmente

- Corrección: ¿resuelve el algoritmo nuestro problema?
- Eficiencia: ¿cuán rápidamente lo hace?
- Si para contar vacas les contamos las patas y dividimos el resultado por cuatro, estamos resolviendo el problema. Pero no de la manera más eficiente.
- El algoritmo de ordenamiento por selección es 3 veces más lento que el MergeSort si tomamos 10 elementos. Es 750 veces más lento si queremos ordenar 1000 elementos.

Corrección de algoritmos. Formalmente

Corrección de algoritmos. Formalmente

- Programa: escritura de un algoritmo en un lenguaje que pueda ser ejecutado por una computadora (Java, Python, &c.)

Corrección de algoritmos. Formalmente

- Programa: escritura de un algoritmo en un lenguaje que pueda ser ejecutado por una computadora (Java, Python, &c.)
- Instancia: una instancia de un programa es un valor de entrada válido. Un programa puede tener numerosas instancias (en general, infinitas)

Corrección de algoritmos. Formalmente

- Programa: escritura de un algoritmo en un lenguaje que pueda ser ejecutado por una computadora (Java, Python, &c.)
- Instancia: una instancia de un programa es un valor de entrada válido. Un programa puede tener numerosas instancias (en general, infinitas)
- Corrección: un programa es correcto (o eficaz) si funciona de la manera esperada para *todas* las instancias de un problema.

¿“Todas” las instancias?

¿"Todas" las instancias?

- ¿Produce el siguiente programa siempre un número primo para $n \geq 0$?

```
1.   algoritmo primo(n : int) {
2.       return n^2 + n + 41;
3.   }
```

¿“Todas” las instancias?

- ¿Produce el siguiente programa siempre un número primo para $n \geq 0$?

```
algoritmo primo(n : int) {
    return n^2 + n + 41;
}
```

- Respuesta: averígüenlo para la próxima clase.

No hay en la tierra una sola página, una sola palabra que [sea simple], ya que todas postulan el universo, cuyo más notorio atributo es la complejidad

J. L. Borges, *El informe de Brodie*

Eficiencia temporal. Principio de invariancia

Eficiencia temporal. Principio de invariancia

Conjetura (Principio de invariancia)

Dado un algoritmo A y dos implementaciones de éste, I_1 e I_2, donde para una entrada de longitud n la primera tarda $T_1(n)$ unidades y la segunda $T_2(n)$ unidades, entonces existe un número real c y un entero n_0 tales que

$$T_1(n) \leq cT_2(n) \quad \textit{para todo} \quad n > n_0$$

Eficiencia temporal. Principio de invariancia

Conjetura (Principio de invariancia)

Dado un algoritmo A y dos implementaciones de éste, I_1 e I_2, donde para una entrada de longitud n la primera tarda $T_1(n)$ unidades y la segunda $T_2(n)$ unidades, entonces existe un número real c y un entero n_0 tales que

$$T_1(n) \leq cT_2(n) \quad \textit{para todo} \quad n > n_0$$

En otras palabras, dos implementaciones distintas del *mismo* algoritmo no difieren sino en una constante multiplicativa.

Mejor caso, peor caso

Mejor caso, peor caso

- Mejor caso: el menor de los costos de todas las instancias de un problema.

Mejor caso, peor caso

- Mejor caso: el menor de los costos de todas las instancias de un problema.
- Caso promedio: el costo promedio de todas las instancias de un problema.

Mejor caso, peor caso

- Mejor caso: el menor de los costos de todas las instancias de un problema.
- Caso promedio: el costo promedio de todas las instancias de un problema.
- Peor caso: el mayor de los costos de todas las instancias de un problema.

Mejor caso, peor caso

- Mejor caso: el menor de los costos de todas las instancias de un problema.
- Caso promedio: el costo promedio de todas las instancias de un problema.
- Peor caso: el mayor de los costos de todas las instancias de un problema.
- Trabajaremos con el peor caso.

Mejor y peor caso: un ejemplo

Considere el siguiente algoritmo, que busca un número x en un arreglo no ordenado T.

```
algoritmo Buscar (T : array[0..n−1] of int, x : int)
int i = 0;
while (i < n) && T[i] ≠ x)
    i++;
if (i < n)
    return i
else
    return −1
```

Mejor y peor caso: un ejemplo

Considere el siguiente algoritmo, que busca un número x en un arreglo no ordenado T.

```
algoritmo Buscar (T : array[0..n − 1] of int, x : int)
int i = 0;
while (i < n) && T[i] ≠ x)
    i + +;
if (i < n)
    return i
else
    return −1
```

- ¿Cuál es el mejor caso?

Mejor y peor caso: un ejemplo

Considere el siguiente algoritmo, que busca un número x en un arreglo no ordenado T.

```
algoritmo Buscar (T : array[0..n − 1] of int, x : int)
int i = 0;
while (i < n) && T[i] ≠ x)
    i++;
if (i < n)
    return i
else
    return −1
```

- ¿Cuál es el mejor caso? $x = T[0]$; sólo se entra una vez al ciclo

Mejor y peor caso: un ejemplo

Considere el siguiente algoritmo, que busca un número x en un arreglo no ordenado T.

```
algoritmo Buscar (T : array[0..n − 1] of int, x : int)
int i = 0;
while (i < n) && T[i] ≠ x)
    i + +;
if (i < n)
    return i
else
    return −1
```

- ¿Cuál es el mejor caso? $x = T[0]$; sólo se entra una vez al ciclo
- ¿Cuál es el peor caso?

Mejor y peor caso: un ejemplo

Considere el siguiente algoritmo, que busca un número x en un arreglo no ordenado T.

```
algoritmo Buscar (T : array[0..n − 1] of int, x : int)
int i = 0;
while (i < n) && T[i] ≠ x)
    i + +;
if (i < n)
    return i
else
    return −1
```

- ¿Cuál es el mejor caso? $x = T[0]$; sólo se entra una vez al ciclo
- ¿Cuál es el peor caso? El número no está o $x = T[n-1]$; se entra n veces al ciclo

Mejor y peor caso: un ejemplo

Considere el siguiente algoritmo, que busca un número x en un arreglo no ordenado T.

```
algoritmo Buscar (T : array[0..n − 1] of int, x : int)
int i = 0;
while (i < n) && T[i] ≠ x)
    i++;
if (i < n)
    return i
else
    return −1
```

- ¿Cuál es el mejor caso? $x = T[0]$; sólo se entra una vez al ciclo
- ¿Cuál es el peor caso? El número no está o $x = T[n-1]$; se entra n veces al ciclo
- ¿Cuál es el caso promedio?

Mejor y peor caso: un ejemplo

Considere el siguiente algoritmo, que busca un número x en un arreglo no ordenado T.

```
algoritmo Buscar (T : array[0..n − 1] of int, x : int)
int i = 0;
while (i < n) && T[i] ≠ x)
    i++;
if (i < n)
    return i
else
    return −1
```

- ¿Cuál es el mejor caso? $x = T[0]$; sólo se entra una vez al ciclo
- ¿Cuál es el peor caso? El número no está o $x = T[n-1]$; se entra n veces al ciclo
- ¿Cuál es el caso promedio? Se ingresa al ciclo $n/2$ veces

Operaciones elementales

Definición (Operaciones elementales)

Una operación elemental *es una operación cuyo tiempo de ejecución máximo puede ser limitado por una constante que depende sólo de la implementación utilizada.*

Operaciones elementales

Definición (Operaciones elementales)

Una operación elemental *es una operación cuyo tiempo de ejecución máximo puede ser limitado por una constante que depende sólo de la implementación utilizada.*

En otras palabras, una operación elemental tiene una duración constante. Esta constante varía según la implementación del lenguaje utilizado.

Cálculo del número de operaciones elementales

Cálculo del número de operaciones elementales

- Para una secuencia de operaciones, la cantidad total de operaciones es la suma de operaciones de la secuencia.

Cálculo del número de operaciones elementales

- Para una secuencia de operaciones, la cantidad total de operaciones es la suma de operaciones de la secuencia.
- Para un condicional, la cantidad de operaciones elementales es 1 más el máximo entre la cantidad de operaciones de ambas ramas de la condición.

Cálculo del número de operaciones elementales

- Para una secuencia de operaciones, la cantidad total de operaciones es la suma de operaciones de la secuencia.
- Para un condicional, la cantidad de operaciones elementales es 1 más el máximo entre la cantidad de operaciones de ambas ramas de la condición.
- Para un ciclo, la cantidad de operaciones elementales es N veces la cantidad de operaciones elementales del bloque de repetición, donde N es la cantidad de repeticiones del ciclo.

Cálculo del número de operaciones elementales

- Para una secuencia de operaciones, la cantidad total de operaciones es la suma de operaciones de la secuencia.
- Para un condicional, la cantidad de operaciones elementales es 1 más el máximo entre la cantidad de operaciones de ambas ramas de la condición.
- Para un ciclo, la cantidad de operaciones elementales es N veces la cantidad de operaciones elementales del bloque de repetición, donde N es la cantidad de repeticiones del ciclo.
- Para una llamada a un método, el número de operaciones elementales es 1 más el número de operaciones elementales de la ejecución del método.

Cálculo del número de operaciones elementales

- Para una secuencia de operaciones, la cantidad total de operaciones es la suma de operaciones de la secuencia.
- Para un condicional, la cantidad de operaciones elementales es 1 más el máximo entre la cantidad de operaciones de ambas ramas de la condición.
- Para un ciclo, la cantidad de operaciones elementales es N veces la cantidad de operaciones elementales del bloque de repetición, donde N es la cantidad de repeticiones del ciclo.
- Para una llamada a un método, el número de operaciones elementales es 1 más el número de operaciones elementales de la ejecución del método.
- ¿Qué pasa en el caso de recurrencia?

La mer, la mer toujours recomencée

(El mar, el mar siempre recomenzado.)

P. Valéry, *Le Cimetière marin*

Algoritmos recursivos

Algoritmos recursivos

- Son algoritmos que se llaman a sí mismos.

Algoritmos recursivos

- Son algoritmos que se llaman a sí mismos.
- Debe haber al menos un *caso base*, que se resuelve sin llamada recursiva.

Algoritmos recursivos

- Son algoritmos que se llaman a sí mismos.
- Debe haber al menos un *caso base*, que se resuelve sin llamada recursiva.
- Para cualquier valor válido de entrada, en cada nuevo llamado la distancia al caso base debe disminuir.

Algoritmos recursivos: un ejemplo (factorial)

Algoritmos recursivos: un ejemplo (factorial)

```
algoritmo factorial(n : int)
   if n == 0
      return 1;
   else
      return n * factorial(n - 1);
}
```

Algoritmos recursivos: un ejemplo (factorial)

```
1.  algoritmo factorial(n : int)
2.      if n == 0
3.          return 1;
4.      else
5.          return n * factorial(n − 1);
6.  }
```

- ¿Hay caso base?

Algoritmos recursivos: un ejemplo (factorial)

```
algoritmo factorial(n : int)
   if n == 0
      return 1;
   else
      return n * factorial(n − 1);
}
```

- ¿Hay caso base?
- ¿Disminuye la distancia al caso base para todos los valores válidos de n?

¿Cómo funciona un algoritmo recursivo?

¿Cómo funciona un algoritmo recursivo?

factorial(5)

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4)

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3)

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2)

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = 2 * factorial(1)

¿Cómo funciona un algoritmo recursivo?

$$\text{factorial}(5) = 5 * \text{factorial}(4)$$

$$\text{factorial}(4) = 4 * \text{factorial}(3)$$

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$\text{factorial}(2) = 2 * \text{factorial}(1)$$

$$\text{factorial}(1)$$

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = 2 * factorial(1)

factorial(1) = 1 * factorial(0)

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = 2 * factorial(1)

factorial(1) = 1 * factorial(0)

factorial(0)

¿Cómo funciona un algoritmo recursivo?

$$\text{factorial}(5) = 5 * \text{factorial}(4)$$

$$\text{factorial}(4) = 4 * \text{factorial}(3)$$

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$\text{factorial}(2) = 2 * \text{factorial}(1)$$

$$\text{factorial}(1) = 1 * \text{factorial}(0)$$

$$\text{factorial}(0) = \mathbf{1}$$

¿Cómo funciona un algoritmo recursivo?

$$\text{factorial}(5) = 5 * \text{factorial}(4)$$

$$\text{factorial}(4) = 4 * \text{factorial}(3)$$

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$\text{factorial}(2) = 2 * \text{factorial}(1)$$

$$\text{factorial}(1) = 1 * \mathbf{1}$$

$$\text{factorial}(0) = \mathbf{1}$$

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = 2 * factorial(1)

factorial(1) = **1**

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = 2 * **1**

factorial(1) = **1**

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = **2**

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * **2**

factorial(2) = **2**

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = **6**

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * **6**

factorial(3) = **6**

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * factorial(4)

factorial(4) = **24**

¿Cómo funciona un algoritmo recursivo?

factorial(5) = 5 * **24**

factorial(4) = **24**

¿Cómo funciona un algoritmo recursivo?

$\text{factorial}(5) = \mathbf{120}$

¿Cómo se construye un algoritmo recursivo?

¿Cómo se construye un algoritmo recursivo?

- ¿Cuál es la estructura del problema? Tenemos

$$\texttt{factorial}(n) = n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$$

¿Cómo se construye un algoritmo recursivo?

- ¿Cuál es la estructura del problema? Tenemos

$$\texttt{factorial}(n) = n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$$

- ¿Cuáles son los casos base? Claramente, $n = 0$ y, opcionalmente, $n = 1$.

¿Cómo se construye un algoritmo recursivo?

- ¿Cuál es la estructura del problema? Tenemos

$$\texttt{factorial}(n) = n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$$

- ¿Cuáles son los casos base? Claramente, $n = 0$ y, opcionalmente, $n = 1$.
- ¿Es una solución para $\texttt{factorial}(n-1)$ útil para obtener una solución de $\texttt{factorial}(n)$? Por supuesto; basta mirar la definición:

$$\begin{aligned} \texttt{factorial}(n) &= n * (n-1) * (n-2) * \ldots * 3 * 2 * 1 \\ &= n * \underbrace{\texttt{factorial}(n-1)}_{\text{recursive part}} \end{aligned}$$

¿Cómo se construye un algoritmo recursivo?

- ¿Cuál es la estructura del problema? Tenemos

$$\texttt{factorial}(n) = n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$$

- ¿Cuáles son los casos base? Claramente, $n = 0$ y, opcionalmente, $n = 1$.
- ¿Es una solución para $\texttt{factorial}(n-1)$ útil para obtener una solución de $\texttt{factorial}(n)$? Por supuesto; basta mirar la definición:

$$\begin{aligned} \texttt{factorial}(n) &= n * (n-1) * (n-2) * \ldots * 3 * 2 * 1 \\ &= n * \underbrace{\texttt{factorial}(n-1)}_{\text{recursive part}} \end{aligned}$$

- Ahora juntemos todo:

```
1.  algoritmo factorial (int n)
2.      if (n == 0||n == 1)
3.          return 1
4.      else
5.          return n * factorial(n − 1)
```

Algoritmos recursivos: un ejemplo (búsqueda binaria)

Algoritmos recursivos: un ejemplo (búsqueda binaria)

```
algorithm binrec (T : array[1..n] of integer, ini, fin, x : integer)
if ini == fin {
   if T[ini] == x
      return ini;
   else
      return −1;
} else {
   mid = (ini + fin + 1)/2
   if x < T[mid]
      return binrec(T, ini, mid − 1, x);
   else
      return binrec(T, mid, fin, x);
}
```

Algoritmos recursivos: un ejemplo (búsqueda binaria)

```
algorithm binrec (T : array[1..n] of integer, ini, fin, x : integer)
if ini == fin {
   if T[ini] == x
      return ini;
   else
      return −1;
} else {
   mid = (ini + fin + 1)/2
   if x < T[mid]
      return binrec(T, ini, mid − 1, x);
   else
      return binrec(T, mid, fin, x);
}
```

- ¿Hay caso base?

Algoritmos recursivos: un ejemplo (búsqueda binaria)

```
algorithm binrec (T : array[1..n] of integer, ini, fin, x : integer)
if ini == fin {
   if T[ini] == x
      return ini;
   else
      return −1;
} else {
   mid = (ini + fin + 1)/2
   if x < T[mid]
      return binrec(T, ini, mid − 1, x);
   else
      return binrec(T, mid, fin, x);
}
```

- ¿Hay caso base?
- ¿Disminuye la distancia al caso base para todos los valores válidos de n?

Las notaciones $\mathcal{O}$ y Θ

Las notaciones $\mathcal{O}$ y Θ

- Notación: $\mathbb{R}^+$ es el conjunto de los reales *no negativos*.

Las notaciones $\mathcal{O}$ y Θ

- Notación: $\mathbb{R}^+$ es el conjunto de los reales *no negativos*.
- Si $f : \mathbb{N} \to \mathbb{R}^+$, entonces $\mathcal{O}(f(n))$ (el "orden" de $f(n)$) es el conjunto de funciones $g : \mathbb{N} \to \mathbb{R}^+$ para las cuales existen constantes c_1 y n_0 tales que

$$0 \leq f(n) \leq c_1 g(n) \text{ para todo } n \geq n_0$$

Las notaciones $\mathcal{O}$ y Θ

- Notación: $\mathbb{R}^+$ es el conjunto de los reales *no negativos*.
- Si $f : \mathbb{N} \to \mathbb{R}^+$, entonces $\mathcal{O}(f(n))$ (el "orden" de $f(n)$) es el conjunto de funciones $g : \mathbb{N} \to \mathbb{R}^+$ para las cuales existen constantes c_1 y n_0 tales que

$$0 \leq f(n) \leq c_1 g(n) \text{ para todo } n \geq n_0$$

- Si $f : \mathbb{N} \to \mathbb{R}^+$, entonces $\Theta(f(n))$ (el "orden exacto" de $f(n)$) es el conjunto de funciones $g : \mathbb{N} \to \mathbb{R}^+$ para las cuales existen constantes c_1, c_2 y n_0 tales que

$$0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \text{ para todo } n \geq n_0$$

¿Y qué significa todo esto?

¿Y qué significa todo esto?

- Se dice que una función $f(n)$ está (por ejemplo) en $\mathcal{O}(n)$, si se puede encontrar alguna constante c_2 tal que a partir de cierto valor $c_2 n$ queda siempre por encima de $f(n)$.

¿Y qué significa todo esto?

- Se dice que una función $f(n)$ está (por ejemplo) en $\mathcal{O}(n)$, si se puede encontrar alguna constante c_2 tal que a partir de cierto valor $c_2 n$ queda siempre por encima de $f(n)$.
- Por ejemplo, $7n + 18 \in \mathcal{O}(n)$, ya que si elegimos $c_2 = 50$, para cualquier valor mayor que cero $50n > 7n + 18$.

¿Y qué significa todo esto?

- Se dice que una función $f(n)$ está (por ejemplo) en $\mathcal{O}(n)$, si se puede encontrar alguna constante c_2 tal que a partir de cierto valor $c_2 n$ queda siempre por encima de $f(n)$.
- Por ejemplo, $7n + 18 \in \mathcal{O}(n)$, ya que si elegimos $c_2 = 50$, para cualquier valor mayor que cero $50n > 7n + 18$.
- Observe que $n^2 \notin \mathcal{O}(n)$; por grande que sea c_2, siempre hay un valor a partir del cual $n^2 > c_2 n$. Por ejemplo, si $c_2 = 1\,000\,000$, a partir de $= 1\,000\,001$ el cuadrado supera a la función lineal.

¿Y qué significa todo esto?

- Se dice que una función $f(n)$ está (por ejemplo) en $\mathcal{O}(n)$, si se puede encontrar alguna constante c_2 tal que a partir de cierto valor $c_2 n$ queda siempre por encima de $f(n)$.
- Por ejemplo, $7n + 18 \in \mathcal{O}(n)$, ya que si elegimos $c_2 = 50$, para cualquier valor mayor que cero $50n > 7n + 18$.
- Observe que $n^2 \notin \mathcal{O}(n)$; por grande que sea c_2, siempre hay un valor a partir del cual $n^2 > c_2 n$. Por ejemplo, si $c_2 = 1\,000\,000$, a partir de $= 1\,000\,001$ el cuadrado supera a la función lineal.
- En la notación $\mathcal{O}$ ignoramos las constantes ($f_1(n) = 3n^2$ y $f_2(n) = 9n^2$ están ambas en $\mathcal{O}(n^2)$) y las funciones de crecimiento menor ($f_3(n) = 5n^2 + 17n + 4$ y $f_4(n) = 9n^2 + 122n - 1$ están ambas en $\mathcal{O}(n^2)$)

¿Y qué significa todo esto?

- Se dice que una función $f(n)$ está (por ejemplo) en $\mathcal{O}(n)$, si se puede encontrar alguna constante c_2 tal que a partir de cierto valor $c_2 n$ queda siempre por encima de $f(n)$.
- Por ejemplo, $7n + 18 \in \mathcal{O}(n)$, ya que si elegimos $c_2 = 50$, para cualquier valor mayor que cero $50n > 7n + 18$.
- Observe que $n^2 \notin \mathcal{O}(n)$; por grande que sea c_2, siempre hay un valor a partir del cual $n^2 > c_2 n$. Por ejemplo, si $c_2 = 1\,000\,000$, a partir de $= 1\,000\,001$ el cuadrado supera a la función lineal.
- En la notación $\mathcal{O}$ ignoramos las constantes ($f_1(n) = 3n^2$ y $f_2(n) = 9n^2$ están ambas en $\mathcal{O}(n^2)$) y las funciones de crecimiento menor ($f_3(n) = 5n^2 + 17n + 4$ y $f_4(n) = 9n^2 + 122n - 1$ están ambas en $\mathcal{O}(n^2)$)
- Lo mismo vale para la notación Θ.

Un ejemplo. Primera parte

Un ejemplo. Primera parte

- El objetivo es mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \Theta(n^3)$.

Un ejemplo. Primera parte

- El objetivo es mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \Theta(n^3)$.
- Para ello, debemos encontrar las constantes c_1, c_2 y n_0 tales que

$$c_2 n^3 \leq 3n^3 + 5n^2 + 7 \leq c_1 n^3 \text{ para todo } n \geq n_0$$

Un ejemplo. Primera parte

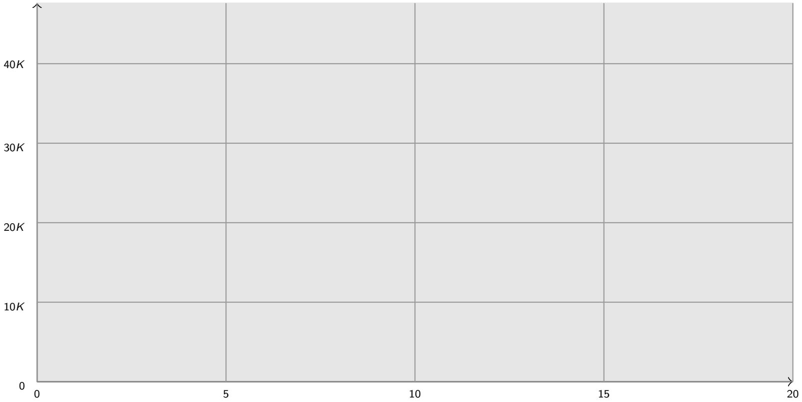
- El objetivo es mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \Theta(n^3)$.
- Para ello, debemos encontrar las constantes c_1, c_2 y n_0 tales que

$$c_2 n^3 \leq 3n^3 + 5n^2 + 7 \leq c_1 n^3 \text{ para todo } n \geq n_0$$

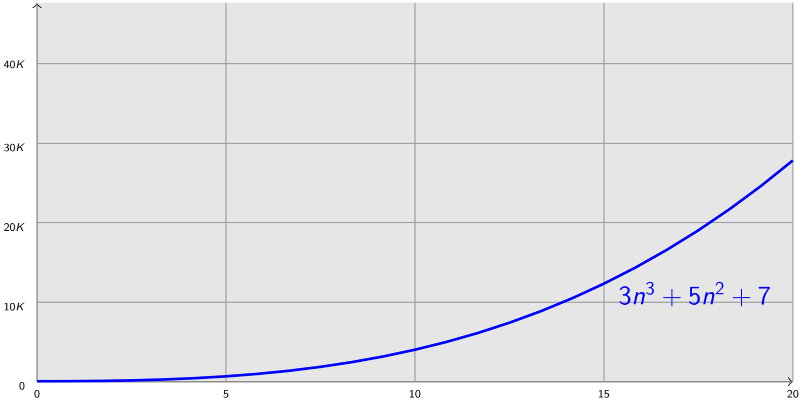
- Por ejemplo, $c_2 = 1$, $c_1 = 5$ y $n_0 = 3$.

Un ejemplo. Segunda parte

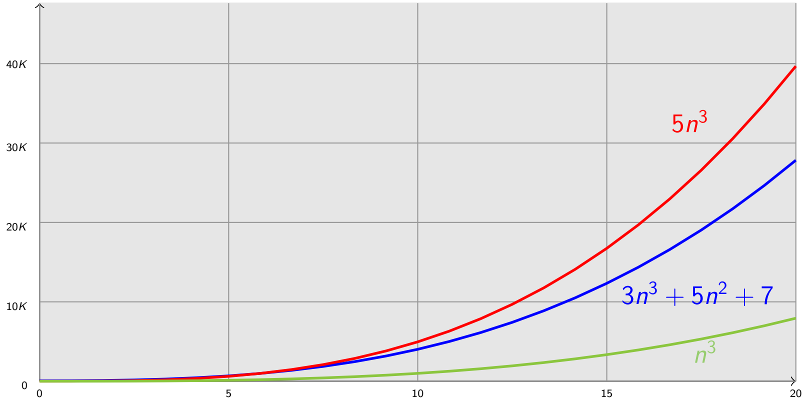
Un ejemplo. Segunda parte



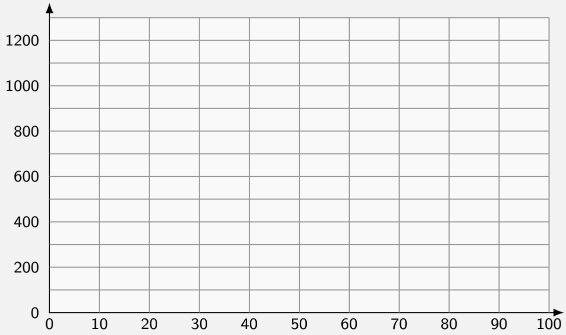
Un ejemplo. Segunda parte



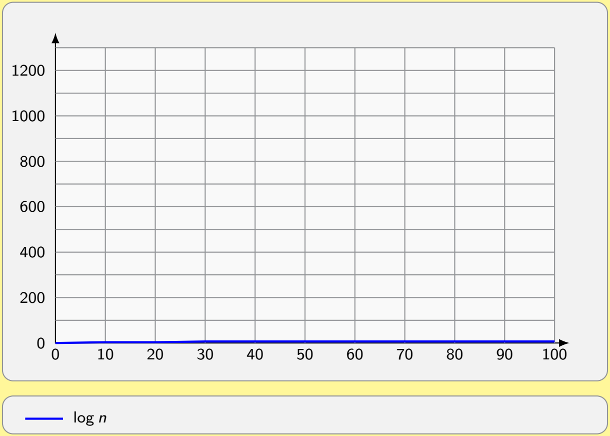
Un ejemplo. Segunda parte



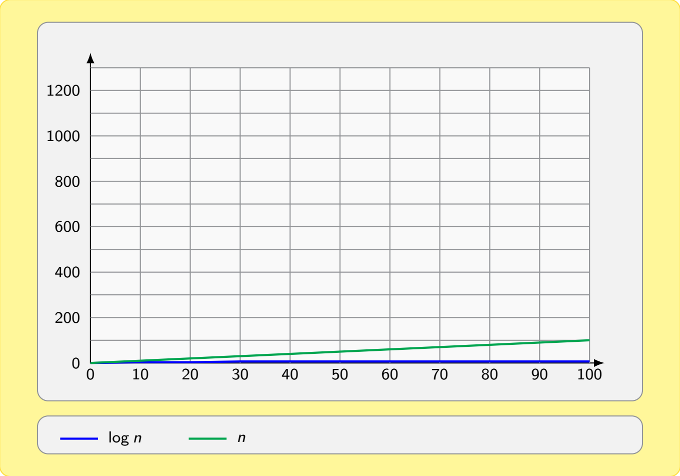
Algunas funciones de crecimiento



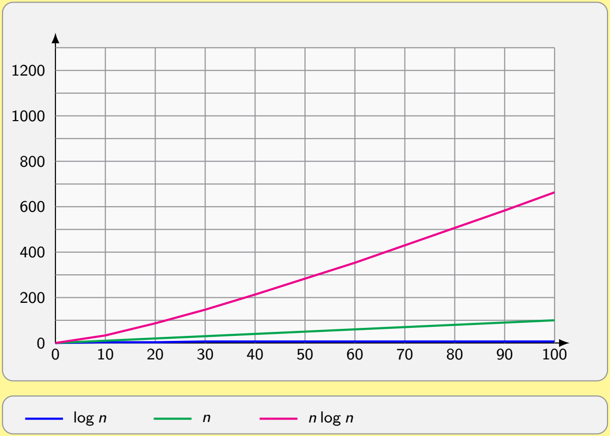
Algunas funciones de crecimiento



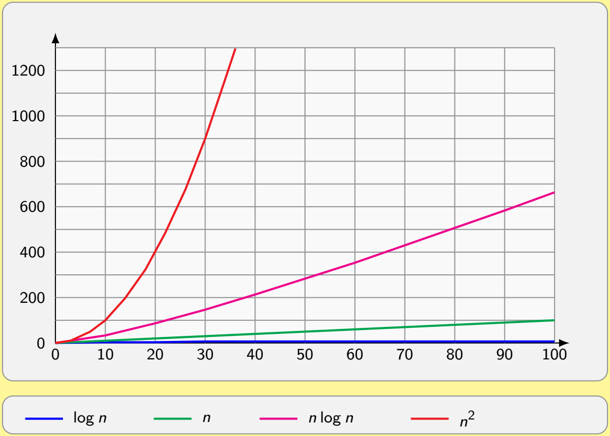
Algunas funciones de crecimiento



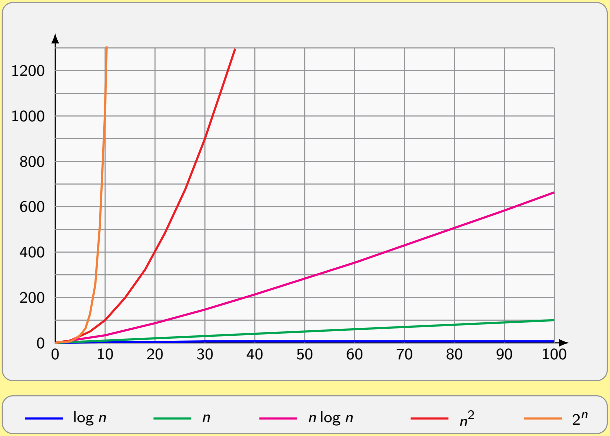
Algunas funciones de crecimiento



Algunas funciones de crecimiento



Algunas funciones de crecimiento



Resolución de recurrencias. Caso de substracción

Resolución de recurrencias. Caso de substracción

- Es una función del tipo

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

Resolución de recurrencias. Caso de substracción

- Es una función del tipo

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

- Donde a es la cantidad de llamadas recursivas (peor caso), b es la cantidad de unidades en que disminuye la entrada en cada paso recursivo (peor caso) y k es el grado del polinomio $p(n)$, que son las sentencias que se ejecutan fuera del llamado recursivo.

Resolución de recurrencias. Caso de substracción

- Es una función del tipo

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

- Donde a es la cantidad de llamadas recursivas (peor caso), b es la cantidad de unidades en que disminuye la entrada en cada paso recursivo (peor caso) y k es el grado del polinomio $p(n)$, que son las sentencias que se ejecutan fuera del llamado recursivo.
- Entonces tenemos:

$$T(n) \in \begin{cases} \Theta\left(n^k\right) & \text{si } a < 1 \\ \Theta\left(n^{k+1}\right) & \text{si } a = 1 \\ \Theta\left(n^k a^{n \div b}\right) & \text{si } a > 1 \end{cases}$$

Resolución de recurrencias. Caso de división

Resolución de recurrencias. Caso de división

- Es una función del tipo

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n/b) + p(n) & \text{si } n \geq b \end{cases}$$

Resolución de recurrencias. Caso de división

- Es una función del tipo

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n/b) + p(n) & \text{si } n \geq b \end{cases}$$

- Donde a es la cantidad de llamadas recursivas (peor caso), b es la cantidad de unidades en que se divide la entrada en cada paso recursivo (peor caso) y k es el grado del polinomio $p(n)$, que son las sentencias que se ejecutan fuera del llamado recursivo.

Resolución de recurrencias. Caso de división

- Es una función del tipo

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n/b) + p(n) & \text{si } n \geq b \end{cases}$$

- Donde a es la cantidad de llamadas recursivas (peor caso), b es la cantidad de unidades en que se divide la entrada en cada paso recursivo (peor caso) y k es el grado del polinomio $p(n)$, que son las sentencias que se ejecutan fuera del llamado recursivo.
- Entonces tenemos:

$$T(n) \in \begin{cases} \Theta\left(n^k\right) & \text{si } a < b^k \\ \Theta\left(n^k \log n\right) & \text{si } a = b^k \\ \Theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Ejemplo de resolución (caso de substracción)

Ejemplo de resolución (caso de substracción)

```
algoritmo factorial(n : int)
    if n = 0
        return 1;
    else
        return n * factorial(n - 1);
}
```

Ejemplo de resolución (caso de substracción)

```
algoritmo factorial(n : int)
    if n = 0
        return 1;
    else
        return n * factorial(n - 1);
}
```

- Aquí tenemos: $a = 1 \quad b = 1 \quad k = 0$

Ejemplo de resolución (caso de substracción)

```
algoritmo factorial(n : int)
    if n = 0
      return 1;
    else
      return n * factorial(n - 1);
}
```

- Aquí tenemos: $a = 1 \quad b = 1 \quad k = 0$
- Por lo tanto: $T(n) \in \Theta(n^{k+1}) = \Theta(n)$

Ejemplo de resolución (caso de división)

Ejemplo de resolución (caso de división)

```
algoritmo binrec (T : array[1..n] of integer, ini, fin, x : integer)
if ini = fin {
    if T[ini] = x
        return ini;
    else
        return 0;
} else {
    k := (ini + fin + 1)/2
    if x<T[k]
        return binrec(T[ini..k − 1], x);
    else
        return binrec(T[k..fin], x);
}
```

Ejemplo de resolución (caso de división)

```
algoritmo binrec (T : array[1..n] of integer, ini, fin, x : integer)
if ini = fin {
   if T[ini] = x
      return ini;
   else
      return 0;
} else {
   k := (ini + fin + 1)/2
   if x < T[k]
      return binrec(T[ini..k − 1], x);
   else
      return binrec(T[k..fin], x);
}
```

- Aquí tenemos: $a = 1 \quad b = 2 \quad k = 0$

Ejemplo de resolución (caso de división)

```
algoritmo binrec (T : array[1..n] of integer, ini, fin, x : integer)
if ini = fin {
   if T[ini] = x
      return ini;
   else
      return 0;
} else {
   k := (ini + fin + 1)/2
   if x<T[k]
      return binrec(T[ini..k − 1], x);
   else
      return binrec(T[k..fin], x);
}
```

- Aquí tenemos: $a = 1 \quad b = 2 \quad k = 0$
- Por lo tanto $T(n) \in \Theta(n^k \log n) = \Theta(\log n)$

Jerarquía (parcial) de casos de complejidad

$\boxed{\mathcal{O}(1)}$ constante

Jerarquía (parcial) de casos de complejidad

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

Jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n \log n)$	quasi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	quasi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n^3)$	cúbica
$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	quasi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Jerarquía (parcial) de casos de complejidad

$\mathcal{O}(2^n)$	exponencial
$\mathcal{O}(n^3)$	cúbica
$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	quasi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n!)$	factorial
$\mathcal{O}(2^n)$	exponencial
$\mathcal{O}(n^3)$	cúbica
$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	quasi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n^n)$	
$\mathcal{O}(n!)$	factorial
$\mathcal{O}(2^n)$	exponencial
$\mathcal{O}(n^3)$	cúbica
$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	quasi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Jerarquía (parcial) de casos de complejidad

$$\begin{array}{ll} \mathcal{O}(n^n) & \\ \mathcal{O}(n!) & \text{factorial} \\ \mathcal{O}(2^n) & \text{exponencial} \end{array}$$

$$\text{tratable}\left\{\begin{array}{ll} \mathcal{O}(n^3) & \text{cúbica} \\ \mathcal{O}(n^2) & \text{cuadrática} \\ \mathcal{O}(n \log n) & \text{quasi lineal} \\ \mathcal{O}(n) & \text{lineal} \\ \mathcal{O}(\log n) & \text{logarítmica} \\ \mathcal{O}(1) & \text{constante} \end{array}\right.$$

Jerarquía (parcial) de casos de complejidad

intratable	$\mathcal{O}(n^n)$	
	$\mathcal{O}(n!)$	factorial
	$\mathcal{O}(2^n)$	exponencial
tratable	$\mathcal{O}(n^3)$	cúbica
	$\mathcal{O}(n^2)$	cuadrática
	$\mathcal{O}(n \log n)$	quasi lineal
	$\mathcal{O}(n)$	lineal
	$\mathcal{O}(\log n)$	logarítmica
	$\mathcal{O}(1)$	constante

Complejidad. Una síntesis

Complejidad. Una síntesis

- Un algoritmo en general tarda más si el tamaño de la entrada n crece.

Complejidad. Una síntesis

- Un algoritmo en general tarda más si el tamaño de la entrada n crece.
- Queremos determinar cómo influye el tamaño de la entrada en el tiempo de ejecución.

Complejidad. Una síntesis

- Un algoritmo en general tarda más si el tamaño de la entrada n crece.
- Queremos determinar cómo influye el tamaño de la entrada en el tiempo de ejecución.
- Las notaciones $\mathcal{O}$ y Θ nos dan una medida aproximada de esta relación.

Complejidad. Una síntesis

- Un algoritmo en general tarda más si el tamaño de la entrada n crece.
- Queremos determinar cómo influye el tamaño de la entrada en el tiempo de ejecución.
- Las notaciones $\mathcal{O}$ y Θ nos dan una medida aproximada de esta relación.
- Ambas notaciones ignoran las constantes: si un algoritmo tarda $4n$ y el otro $7n$, ambos están en el mismo orden de complejidad, $\mathcal{O}(n)$ (o $\Theta(n)$).

Complejidad. Una síntesis

- Un algoritmo en general tarda más si el tamaño de la entrada n crece.
- Queremos determinar cómo influye el tamaño de la entrada en el tiempo de ejecución.
- Las notaciones $\mathcal{O}$ y Θ nos dan una medida aproximada de esta relación.
- Ambas notaciones ignoran las constantes: si un algoritmo tarda $4n$ y el otro $7n$, ambos están en el mismo orden de complejidad, $\mathcal{O}(n)$ (o $\Theta(n)$).
- Ambas notaciones ignoran los términos menos significativos: si un algoritmo tarda $2n^2 + 3n - 5$ y el otro $7n^2 + 18n + 144$, nos quedamos con los términos más importantes, o sea con $2n^2$ y $7n^2$ y, por lo que se dijo arriba, ambos están en el mismo orden de complejidad, $\mathcal{O}(n^2)$ (o $\Theta(n^2)$).

1. Introducción. Paisaje general
2. Algoritmos
3. Complejidad temporal
   - Recurrencia
   - Notaciones asintóticas
   - P & NP
4. Ejercicios propuestos

Problemas polinomiales y exponenciales

Problemas polinomiales y exponenciales

Complejidad polinomial

Heap Sort
Merge Sort
Bubble Sort
Búsqueda binaria
Producto de matrices
Factorial
Agregar a un Heap
Comparación de conjuntos
Producto escalar

Problemas polinomiales y exponenciales

Complejidad polinomial

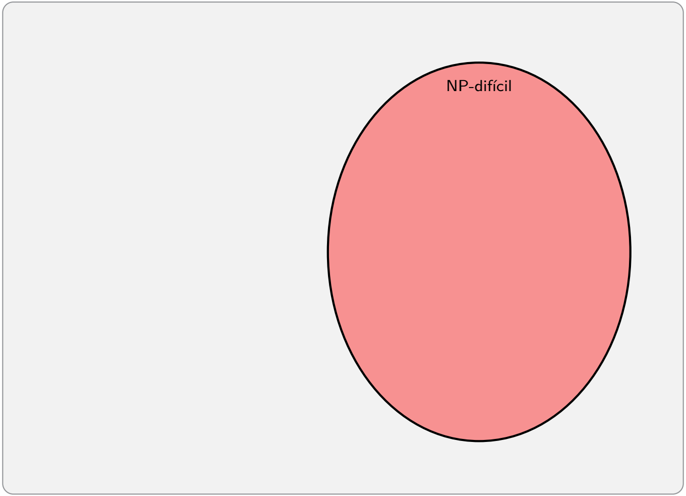
Heap Sort
Merge Sort
Bubble Sort
Búsqueda binaria
Producto de matrices
Factorial
Agregar a un Heap
Comparación de conjuntos
Producto escalar

Exponential Complexity

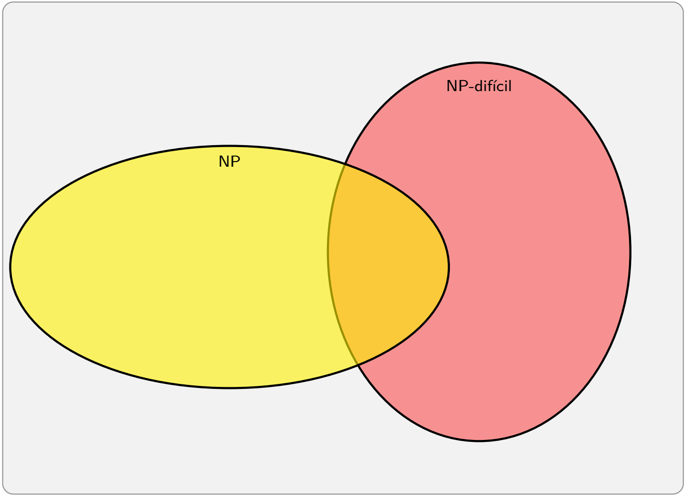
Ciclo Hamiltoniano
TSP
Mochila 0-1
SAT
Suma de subconjuntos
Logaritmo Discreto
Camarillas
Cobertura de vértices
Conjunto independiente

Algunas clases de complejidad

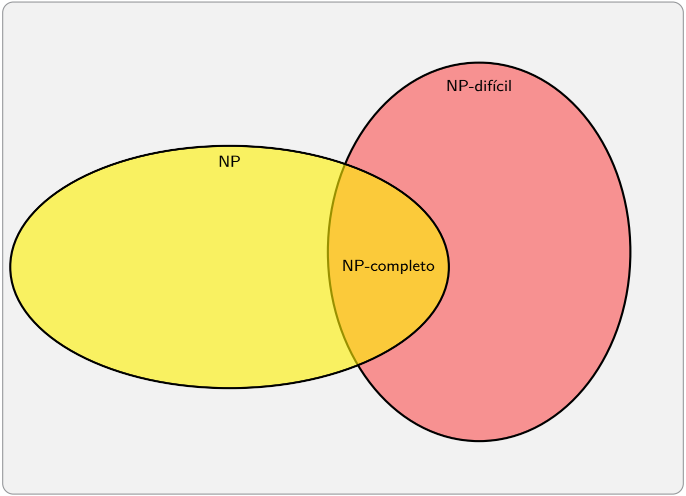
Algunas clases de complejidad



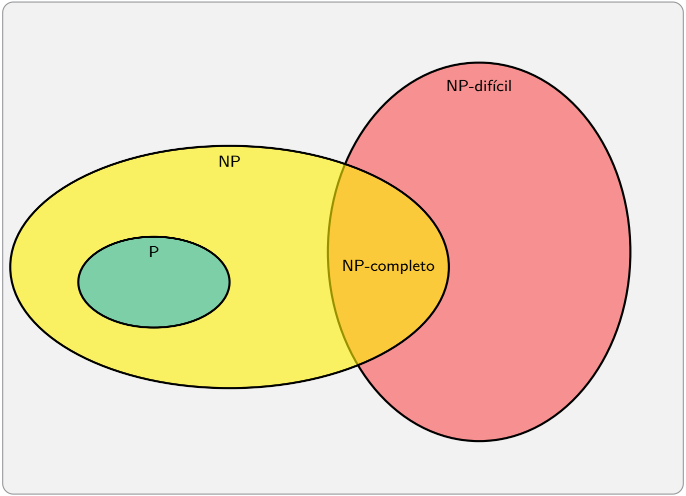
Algunas clases de complejidad



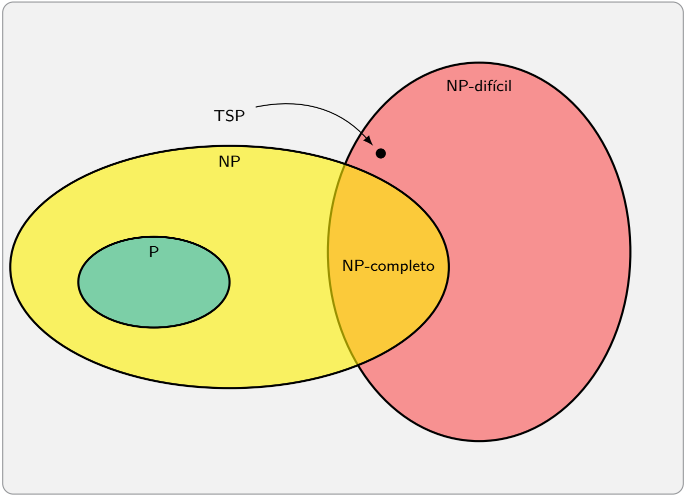
Algunas clases de complejidad



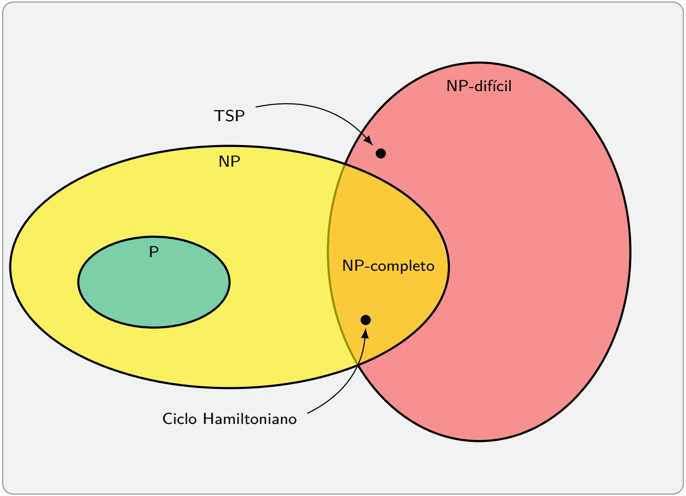
Algunas clases de complejidad



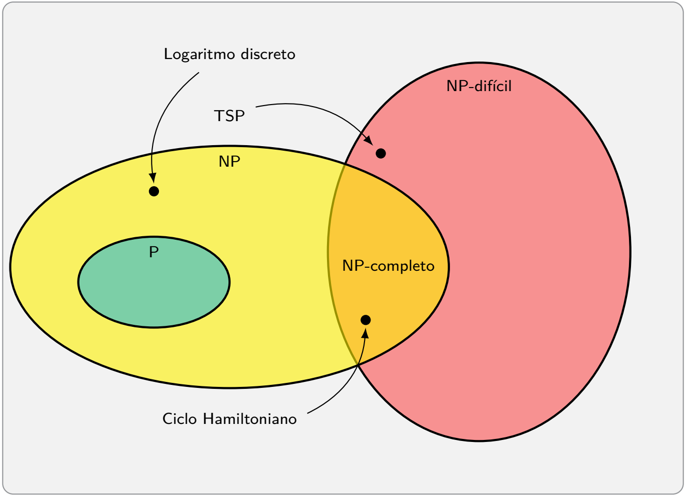
Algunas clases de complejidad



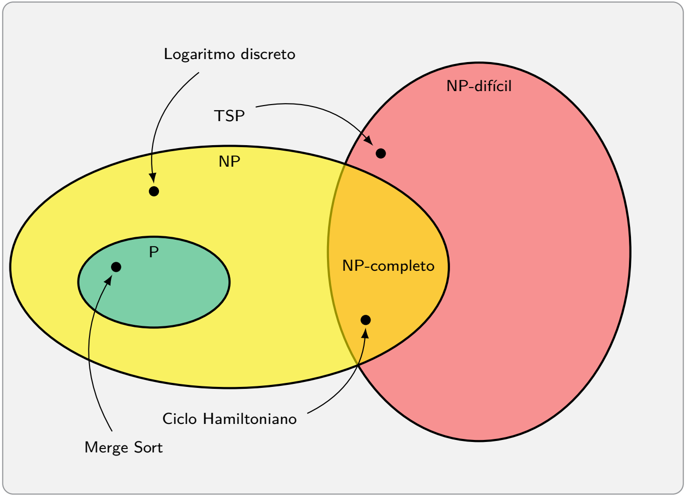
Algunas clases de complejidad



Algunas clases de complejidad



Algunas clases de complejidad



Ejercicios propuestos 1

1. "Este algoritmo se ejecuta por lo menos en $\mathcal{O}(n^2)$". ¿Qué opinión le merece esta frase?
2. El algoritmo A_1 resuelve un problema en n^2 días y el algoritmo A_2 lo resuelve en n^3 segundos. ¿A partir de qué tamaño de la entrada el algoritmo A_1 supera al algoritmo A_2? ¿Cuánto demora la solución en este caso?
3. ¿Es cierto que $n^2 \in \mathcal{O}(n^3)$? ¿Y que $n^2 \in \Theta(n^3)$?
4. Calcule la complejidad de los algoritmos que se dan en las siguientes diapositivas.

Algoritmos para los problemas propuestos — Algoritmo A

```
Algoritmo AlgoA(n : int)
    if n ≥ 2
        bool a ← getValue(n);    // getValue ∈ Θ(n^2)
    if a {
        print (n);
        AlgoA(n/2);
    } else {
        AlgoA(n/2);
    }
```

Algoritmos para los problemas propuestos — Algoritmo B

```
Algoritmo AlgoB(n : int)
    if n ≥ 2 {
        bool a ← getValue(n)        // getValue ∈ Θ(1)
        if a {
            CalculateCost(n);       // CalculateCost ∈ Θ(n^2)
            AlgoB(n/2);
        } else {
            AlgoB(n/2);
        }
    } else {
        AlgoB(n/2);
    }
```

Algoritmos para los problemas propuestos — Algoritmo C

```
Algoritmo AlgoC(n : int)
    if n ≥ 2 {
        bool a ← getValue(n);          // getValue ∈ Θ(1)
        if a {
            CalculateCost(n);          // CalculateCost ∈ Θ(n^2)
            AlgoC(n/2);
        } else {
            AlgoC(n/2);
        }
    } else {
        AssessRisk(n);                 // AssessRisk ∈ Θ(n)
        AlgoC(n/2)
        CalculateCost(n);              // CalculateCost ∈ Θ(n^2)
        AlgoC(n/2);
    }
```

Ejercicios propuestos 2

5. Suponga que tiene que elegir entre los siguientes algoritmos para resolver un problema:
    - El algoritmo A resuelve el problema dividiéndolo en cinco subproblemas de la mitad de tamaño, resolviendo recursivamente cada subproblema y combinando las soluciones en tiempo lineal.
    - El algoritmo B resuelve un problema de tamaño n resolviendo recursivamente dos problemas de tamaño $n-1$ y combinándolos en tiempo constante.
    - El algoritmo C resuelve un problema de tamaño n dividiéndolo en nueve subproblemas de tamaño $n/3$, resolviendo recursivamente cada subproblema y combinando las soluciones en tiempo cuadrático.

    ¿Cuál es la complejidad de cada uno de estos algoritmos y cuál elegiría?