Proceso de Admisión 2021

Curso de Programación Básica

Cinvestav
Tamaulipas

Dr. Miguel Morales Sandoval

Tema:

1. Análisis y diseño de algoritmos
   - 1.1 Algoritmos
   - 1.2 Análisis de algoritmos
   - 1.3 Complejidad algorítmica
   - 1.4 Notación asintótica
   - 1.5 Ejercicios
   - 1.6 Complejidad de principales algoritmos

Repaso sobre **Algoritmos**

Algoritmos

1. ¿Qué es un algoritmo?

Algoritmos

1. ¿Qué es un algoritmo?
    - Secuencia de pasos no ambiguos para **resolver un problema**. En computación, el algoritmo son intrucciones de cálculo, almacenamiento o lectura de variables.

Algoritmos

1. ¿Qué es un algoritmo?
    - Secuencia de pasos no ambiguos para **resolver un problema**. En computación, el algoritmo son intrucciones de cálculo, almacenamiento o lectura de variables.
    - Requiere de datos de **entrada**, los cuales se usan en los pasos del algoritmo
    - Produce una **salida** o resultado después de la ejecución de cada uno de los pasos
    - Usa determinados recursos de **cómputo** o de **almacenamiento** durante la ejecución de cada paso el algoritmo

Algoritmos

1. ¿Qué es un algoritmo?
    - Secuencia de pasos no ambiguos para **resolver un problema**. En computación, el algoritmo son intrucciones de cálculo, almacenamiento o lectura de variables.
    - Requiere de datos de **entrada**, los cuales se usan en los pasos del algoritmo
    - Produce una **salida** o resultado después de la ejecución de cada uno de los pasos
    - Usa determinados recursos de **cómputo** o de **almacenamiento** durante la ejecución de cada paso el algoritmo

Algoritmos

¿Es el siguiente un algoritmo?

```
1  1 Input: some data
   Output: some result
2  continue ← true;
3  i ← 0;
4  while continue do
5  |   i ← ∗ = 2;
6  |   doSomeProcess(i);
7  |   if i%2 == 0 then
8  |   |   break;
9  |   end
10 end
```

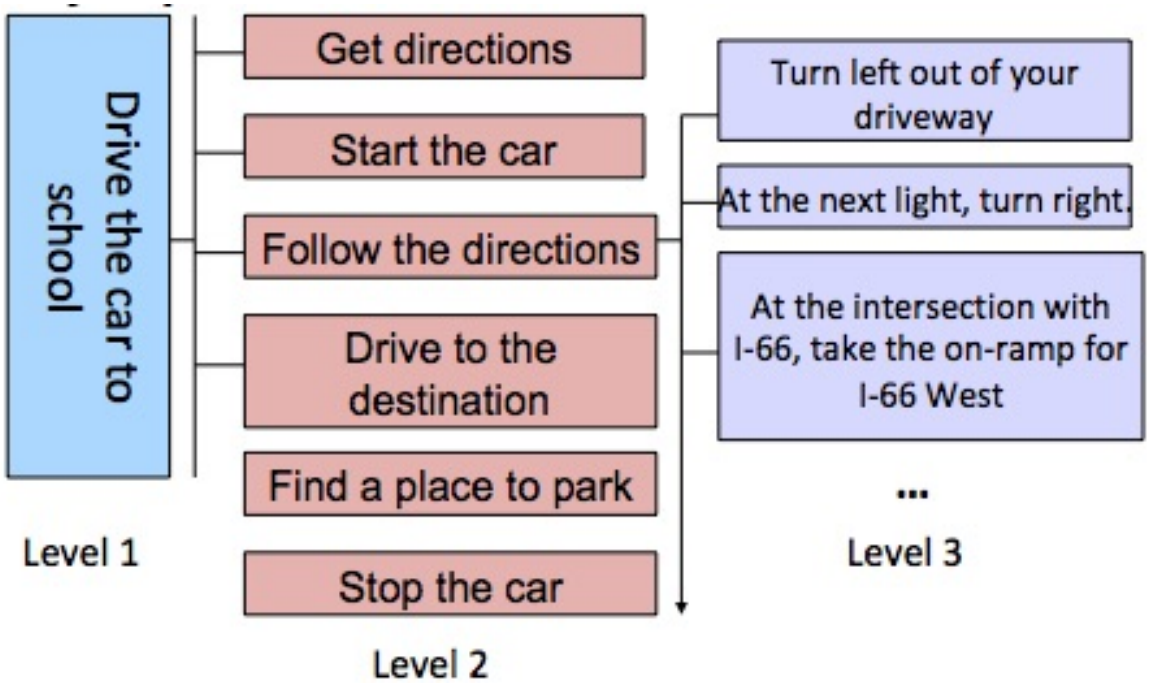
Secuencia de pasos 1: ¿es un algoritmo?

Algoritmos

Características de un buen algoritmo

1. **Precisión** – los pasos que conforman el algoritmo están claramente definidos (no ambiguos).
2. **Unicidad** – el resultado de cada paso en el algoritmo está únicamente definido por los datos de entrada de ese paso y por los resultados de los pasos anteriores.
3. **Finitidad** – el algoritmo termina después de que se han ejecutado un número finito de pasos.
4. **Generalidad** – el algoritmo es aplicable a diversos conjuntos de entrada.

Algoritmos



Especificación de un algoritmo a distintos niveles de detalle.

Construcción y validación de un algoritmo

1. Analizar el problema y desarrolla la especificación del mismo (determinar los datos de entrada, resultado, y restriciones).

Algoritmos

Construcción y validación de un algoritmo

1. Analizar el problema y desarrolla la especificación del mismo (determinar los datos de entrada, resultado, y restriciones).
2. Diseñar la solución (crear el o los algoritmos requeridos (determinar los pasos) y verificar que el diseño es correcto).

Algoritmos

Construcción y validación de un algoritmo

1. Analizar el problema y desarrolla la especificación del mismo (determinar los datos de entrada, resultado, y restriciones).
2. Diseñar la solución (crear el o los algoritmos requeridos (determinar los pasos) y verificar que el diseño es correcto).
3. Implementar el algoritmo (crear el programa) y verifica que la escritura del programa es correcta.

Algoritmos

Construcción y validación de un algoritmo

1. Analizar el problema y desarrolla la especificación del mismo (determinar los datos de entrada, resultado, y restriciones).
2. Diseñar la solución (crear el o los algoritmos requeridos (determinar los pasos) y verificar que el diseño es correcto).
3. Implementar el algoritmo (crear el programa) y verifica que la escritura del programa es correcta.
4. Verificar el programa, mediante depuración y realización de pruebas para asegurar que el algoritmo es **correcto** y **completo**, bajo "todos"los casos de prueba.

Algoritmos

Un algoritmo es **correcto/válido** si resuelve el problema computacional para el cual fue diseñado. Para cada entrada, produce la salida deseada (correcta).

Algoritmos

Un algoritmo es **correcto/válido** si resuelve el problema computacional para el cual fue diseñado. Para cada entrada, produce la salida deseada (correcta).

Forma manual de verificar la correctitud de un algoritmo:

1. Seleccionar un conjunto de datos de entrada *DE*
2. Ejecutar el algoritmo con *DE*
3. Verificar el resultado R

Algoritmos

¿Cuántos *DE*s podemos pasarle a un algoritmo?

Algoritmos

¿Cuántos *DE*s podemos pasarle a un algoritmo?

Cada *DE* define **una instancia** del problema.

Algoritmos

¿Cuántos *DE*s podemos pasarle a un algoritmo?
Cada *DE* define **una instancia** del problema.

Un algoritmo **es completo** si garantiza retornar una respuesta correcta para cualquier entrada *DE*, o si la respuesta no existe, lo informa.

Algoritmos

¿Cuántos *DE*s podemos pasarle a un algoritmo?
Cada *DE* define **una instancia** del problema.

Un algoritmo **es completo** si garantiza retornar una respuesta correcta para cualquier entrada *DE*, o si la respuesta no existe, lo informa.

¿Cuántas soluciones podemos dar a un problema? –> ¿Cuántos algoritmos se pueden proponer para resolver un problema? –> ¿Cuál de todos es el mejor?

Algoritmos

¿Cuántos *DE*s podemos pasarle a un algoritmo?
Cada *DE* define **una instancia** del problema.

Un algoritmo **es completo** si garantiza retornar una respuesta correcta para cualquier entrada *DE*, o si la respuesta no existe, lo informa.

¿Cuántas soluciones podemos dar a un problema? –> ¿Cuántos algoritmos se pueden proponer para resolver un problema? –> ¿Cuál de todos es el mejor?

El de menor complejidad, el que cueste menos.

Análisis de algoritmos

El **análisis de algoritmos** consiste en determinar la complejidad de la solución propuesta (algoritmo) a un problema dado.
¿Cuántas operaciones básicas requiere el algoritmo? –> **costo en tiempo**
¿Cuánta memoria requiere el algoritmo? –> **costo en espacio**

Análisis de algoritmos

1. ¿Los algoritmos creados e implementados se pueden ejecutar en una computadora en tiempo razonable? **Complejidad temporal**
2. ¿El espacio de memoria requerido por los algoritmos creados está disponible en la computadora donde se ejecutarán los mismos? **Complejidad espacial**

Análisis de algoritmos

1. ¿Los algoritmos creados e implementados se pueden ejecutar en una computadora en tiempo razonable? **Complejidad temporal**
2. ¿El espacio de memoria requerido por los algoritmos creados está disponible en la computadora donde se ejecutarán los mismos? **Complejidad espacial**

La respuesta a estas preguntas, intuitivamente, depende del **tamaño de la entrada** o **instancia del problema**.

Análisis de algoritmos

1. ¿Cómo calculamos la **complejidad temporal** de un algoritmo?

Análisis de algoritmos

1. ¿Cómo calculamos la **complejidad temporal** de un algoritmo?
   - 1.1 Determinar en el algoritmo el número de pasos
   - 1.2 Determinar en el algoritmo el número de operaciones básicas
   - 1.3 Determinar en el algoritmo el número de ciclos de procesador

Análisis de algoritmos

1. ¿Cómo calculamos la **complejidad temporal** de un algoritmo?
   - 1.1 Determinar en el algoritmo el número de pasos
   - 1.2 Determinar en el algoritmo el número de operaciones básicas
   - 1.3 Determinar en el algoritmo el número de ciclos de procesador
2. ¿y la **complejidad espacial**?

Análisis de algoritmos

1. ¿Cómo calculamos la **complejidad temporal** de un algoritmo?
   - 1.1 Determinar en el algoritmo el número de pasos
   - 1.2 Determinar en el algoritmo el número de operaciones básicas
   - 1.3 Determinar en el algoritmo el número de ciclos de procesador
2. ¿y la **complejidad espacial**?
   - 2.1 Determinar número de variables (arreglos, estructuras, listas)
   - 2.2 Determinar el espacio utilizado por las variables en el algoritmo
   - 2.3 Determinar cantidad de datos almacenados en disco

Análisis de algoritmos

La complejidad (espacial y temporal) de un algoritmo se calculan **en función** de los datos de entrada (**instancia del problema**)

Análisis de algoritmos

Un **algoritmo** debería **ser el mejor posible**: se ejecuta **más rápido** que otros algoritmos que resuelven el mismo problema o usa **menos recursos de memoria**. Es decir, el algoritmo debería tener la **complejidad más baja posible**.

¿Cuándo un algoritmo es mejor que otro?, ¿Cómo lo sabemos?

Análisis de algoritmos

Un **algoritmo** debería **ser el mejor posible**: se ejecuta **más rápido** que otros algoritmos que resuelven el mismo problema o usa **menos recursos de memoria**. Es decir, el algoritmo debería tener la **complejidad más baja posible**.

¿Cuándo un algoritmo es mejor que otro?, ¿Cómo lo sabemos?

Comparando sus complejidades

Análisis de algoritmos

1. **Complejidad temporal**
   1.1 En el análisis de algoritmos, no se hace una comparación de tiempos de ejecución entre algoritmos para determinar cuál es el mejor.

Análisis de algoritmos

1. **Complejidad temporal**
   - 1.1 En el análisis de algoritmos, no se hace una comparación de tiempos de ejecución entre algoritmos para determinar cuál es el mejor.
   - 1.2 El tiempo de ejecución es variable, depende de muchos factores.

Análisis de algoritmos

1. **Complejidad temporal**
   - 1.1 En el análisis de algoritmos, no se hace una comparación de tiempos de ejecución entre algoritmos para determinar cuál es el mejor.
   - 1.2 El tiempo de ejecución es variable, depende de muchos factores.
   - 1.3 El análisis es más formal, matemático (no depende de la implementación)

Análisis de algoritmos

1. **Complejidad temporal**
   - 1.1 En el análisis de algoritmos, no se hace una comparación de tiempos de ejecución entre algoritmos para determinar cuál es el mejor.
   - 1.2 El tiempo de ejecución es variable, depende de muchos factores.
   - 1.3 El análisis es más formal, matemático (no depende de la implementación)

El análisis de algoritmos se enfoca más en deteminar la complejidad temporal de los algoritmos. A menos que se exprese lo contrario, por complejidad de un algoritmo nos referiremos a su **complejidad temporal**.

Análisis de algoritmos

Considere el problema de cálculo de una exponenciación: **Dado** n **un número entero y** x **un número real, calcular** x^n. Considere los siguientes dos algoritmos que resuelven el problema anterior:

Input: n, x
Output: x^n

```
1 if n > 0 then
2 |   regresar x * x^(n-1)
3 end
4 else
5 |   regresar 1
6 end
```

Input: n, x
Output: x^n

```
1 if n es par then
2 |   regresar (x^(n/2))^2
3 end
4 else
5 |   regresar x * (x^(n/2))^2
6 end
```

Análisis de algoritmos

Considere el problema de cálculo de una exponenciación: **Dado** n **un número entero y** x **un número real, calcular** x^n. Considere los siguientes dos algoritmos que resuelven el problema anterior:

Input: n, x
Output: x^n

```
1 if n > 0 then
2 |   regresar x * x^(n-1)
3 end
4 else
5 |   regresar 1
6 end
```

Input: n, x
Output: x^n

```
1 if n es par then
2 |   regresar (x^(n/2))^2
3 end
4 else
5 |   regresar x * (x^(n/2))^2
6 end
```

¿Cuál de los dos algoritmos **es mejor**? ¿Lo podemos deducir a simple vista?

Receso de 5 minutos

Análisis de algoritmos

La **Complejidad Algorítmica** es una estimación del número de pasos en un algoritmo, el cual depende del tamaño de los datos de entrada.

Análisis de algoritmos

La **Complejidad Algorítmica** es una estimación del número de pasos en un algoritmo, el cual depende del tamaño de los datos de entrada.

¿Cuál es el tamaño de la entrada en el siguiente algoritmo?

Input: n, x
Output: x^n

```
1 if n es par then
2 |   regresar (x^{n/2})^2
3 end
4 else
5 |   regresar x * (x^{n/2})^2
6 end
```

Análisis de algoritmos

Análisis de la Complejidad de un Algoritmo

1. Sea n el tamaño de la entrada del algoritmo
2. En un algoritmo, las operaciones básicas son una unidad de pasos de ejecución (asignaciones, cálculos aritméticos, etc). Estas operaciones, en cualquier arquitectura de cómputo tendrán un costo aunque variable, finito. Dicha variación debido a la tecnología usada estará determinada por una constante.
3. Contabilizar el número de operaciones básicas en función de n

Análisis de la Complejidad Temporal: Ejemplo

Considere el siguiente problema. **Dado un arreglo A de n elementos, y un valor v, determinar si $v \in A$.**

1. Proponga un algoritmo que resuelva el problema anterior y contabilice cuántas operaciones requiere.

Análisis de la Complejidad Temporal: Ejemplo

Considere el siguiente problema. **Dado un arreglo A de n elementos, y un valor v, determinar si $v \in A$.**

1. Proponga un algoritmo que resuelva el problema anterior y contabilice cuántas operaciones requiere.

```
buscar(A, n, v){
    for each (v1 ∈ A)
        if (v1 == v) return true
return false
}
```

Análisis de la Complejidad Temporal: Ejemplo

Considere el siguiente problema. **Dado un arreglo A de n elementos, y un valor v, determinar si $v \in A$.**

1. Sea $T(n)$ el número de operaciones del algoritmo *buscar(A, n, v)*, en función del tamaño de los datos de entrada n.
2. $T(n) = n$ es la complejidad del algoritmo *buscar(A, n, v)*, en el **peor caso**.
3. Si graficamos $T(n)$ para diferentes valores de A, ¿qué tipo de gráfica obtenemos?

Análisis de la Complejidad Temporal: Ejemplo

Considere el siguiente problema. **Dado un arreglo A de n elementos, y un valor v, determinar si $v \in A$.**

1. Sea $T(n)$ el número de operaciones del algoritmo *buscar(A, n, v)*, en función del tamaño de los datos de entrada n.
2. $T(n) = n$ es la complejidad del algoritmo *buscar(A, n, v)*, en el **peor caso**.
3. Si graficamos $T(n)$ para diferentes valores de A, ¿qué tipo de gráfica obtenemos?
4. La gráfica es una LÍNEA con pendiente 45 grados.
5. El comportamiento LINEAL del algoritmo *buscar(A, n, v)* será el mismo, independientemente de la velocidad de procesador del equipo donde se ejecute el algoritmo.

Análisis de algoritmos

Piense y dé un ejemplo de un algoritmo, cuya número de operaciones $T(n) = 1$, esto es, independientemente del tamaño de la entrada, el número de operaciones sea 1.

Análisis de algoritmos

Piense y dé un ejemplo de un algoritmo, cuya número de operaciones $T(n) = 1$, esto es, independientemente del tamaño de la entrada, el número de operaciones sea 1.

```
encontrarCentro(A, n){
        return A[n/2]
}
```

Análisis de algoritmos

Piense y dé un ejemplo de un algoritmo, cuya número de operaciones $T(n) = 1$, esto es, independientemente del tamaño de la entrada, el número de operaciones sea 1.

```
encontrarCentro(A, n){
        return A[n/2]
}
```

La complejidad de *encontrarCentro(A, n)* es $T(n) = 1$. ¿Cómo se vería la gráfica de $T(n)$ para distintos valores de n?

Análisis de algoritmos

Piense y dé un ejemplo de un algoritmo, cuya número de operaciones $T(n) = 1$, esto es, independientemente del tamaño de la entrada, el número de operaciones sea 1.

```
encontrarCentro(A, n){
        return A[n/2]
}
```

La complejidad de *encontrarCentro(A, n)* es $T(n) = 1$. ¿Cómo se vería la gráfica de $T(n)$ para distintos valores de n?
En este caso que $T(n) = 1$, decimos que el algoritmo tiene una complejidad CONSTANTE. Lo mismo pasaría si $T(n) = c$, para cualquier otro número de operaciones constante c.

Análisis de algoritmos

Si un algoritmo A realiza $T_A(n) = c_1$ operaciones y un algoritmo B realiza $T_B(n) = c_2$ operaciones, ambos algoritmos tendrán una **complejidad CONSTANTE**, ¿porqué?

Análisis de algoritmos

Si un algoritmo A realiza $T_A(n) = c_1 n$ operaciones y un algoritmo B realiza $T_B(n) = c_2 n$ operaciones, ambos algoritmos tendrán una **complejidad LINEAL**, ¿porqué?

Análisis de algoritmos

Distintos algoritmos pertenecen a la **misma familia de acuerdo con su complejidad**, medida por el número de operaciones que realizan. Una de estas familias es la de complejidad CONSTANTE.

Análisis de algoritmos

Distintos algoritmos pertenecen a la **misma familia de acuerdo con su complejidad**, medida por el número de operaciones que realizan. Una de estas familias es la de complejidad CONSTANTE.

Existen otras, como la complejidad LINEAL, LOGARÍTMICA, CUADRÁTICA, CÚBICA, POLINOMIAL, EXPONENCIAL.

Análisis de algoritmos

Ejemplo:

Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	2
for (*i* =1; i<n; i++)	2*n*
(i=1 once, i<n n times, i++ (n-1) times)	
if *A*[*i*] > *currentMax* **then**	2(*n* - 1)
currentMax ← *A*[*i*]	2(*n* - 1)
return *currentMax*	1
Total	6*n* -1

Análisis de algoritmos

1. `arrayMax(A,n)` se ejecuta en $T(n) = 6n - 1$ operaciones básicas, en **el peor caso**.

Análisis de algoritmos

1. `arrayMax(A,n)` se ejecuta en $T(n) = 6n - 1$ operaciones básicas, en **el peor caso**.
   1.1 Si graficamos la función $T(n)$ para distintos valores de n, ¿cómo se verá la gráfica?.

Análisis de algoritmos

1. `arrayMax(A,n)` se ejecuta en $T(n) = 6n - 1$ operaciones básicas, en **el peor caso**.
   - 1.1 Si graficamos la función $T(n)$ para distintos valores de n, ¿cómo se verá la gráfica?.
   - 1.2 El tasa de crecimiento del tiempo de ejecución dado por $T(n)$ es una propiedad intrínseca del algoritmo `arrayMax()`, independientemente de la computadora donde se ejecute.

Análisis de algoritmos

1. `arrayMax(A,n)` se ejecuta en $T(n) = 6n - 1$ operaciones básicas, en **el peor caso**.
   - 1.1 Si graficamos la función $T(n)$ para distintos valores de n, ¿cómo se verá la gráfica?.
   - 1.2 El tasa de crecimiento del tiempo de ejecución dado por $T(n)$ es una propiedad intrínseca del algoritmo `arrayMax()`, independientemente de la computadora donde se ejecute.
2. ¿La complejidad del algoritmo `buscar(A,n,v)` es la misma que la del algoritmo `arrayMax(A,n)`?

Complejidad algorítmica

La **Complejidad Algorítmica** se mide mediante **notación asintótica** (comportamiento del crecimiento de la función para diversos valores de n, cuando n tiende a ser muy grande).

Complejidad algorítmica

La **Complejidad Algorítmica** se mide mediante **notación asintótica** (comportamiento del crecimiento de la función para diversos valores de n, cuando n tiende a ser muy grande).

En otras palabras, la **notación asintótica** indica qué tan rápido crece $T(n)$ respecto a n.

Complejidad algorítmica

La **Complejidad Algorítmica** se mide mediante **notación asintótica** (comportamiento del crecimiento de la función para diversos valores de n, cuando n tiende a ser muy grande).

En otras palabras, la **notación asintótica** indica qué tan rápido crece $T(n)$ respecto a n.

Permite agrupar funciones $T(n)$ asociadas a complejidad de algoritmos dentro de una **misma clase** (CONSTANTE, LINEAL, etc) .

Complejidad algorítmica

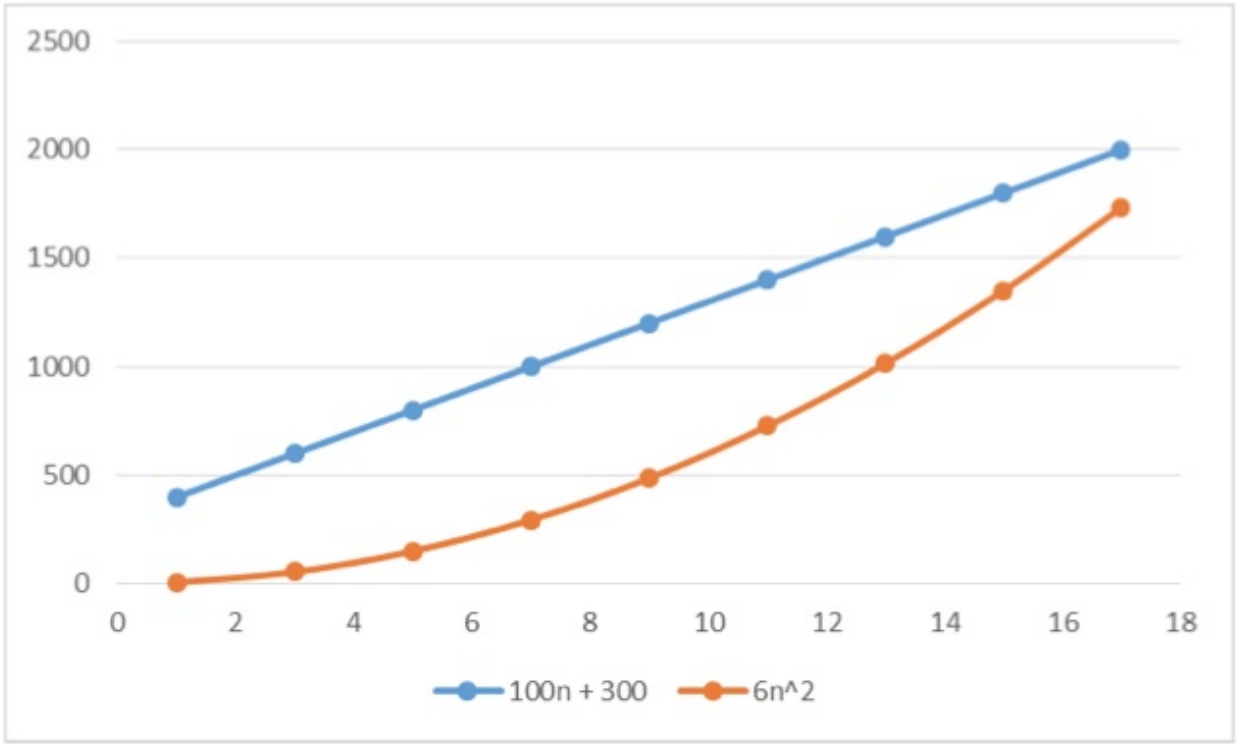
¿Qué función crece más rápido?

a) $T_1(n) = 100n + 300$

b) $T_2(n) = 6n^2 + 10$

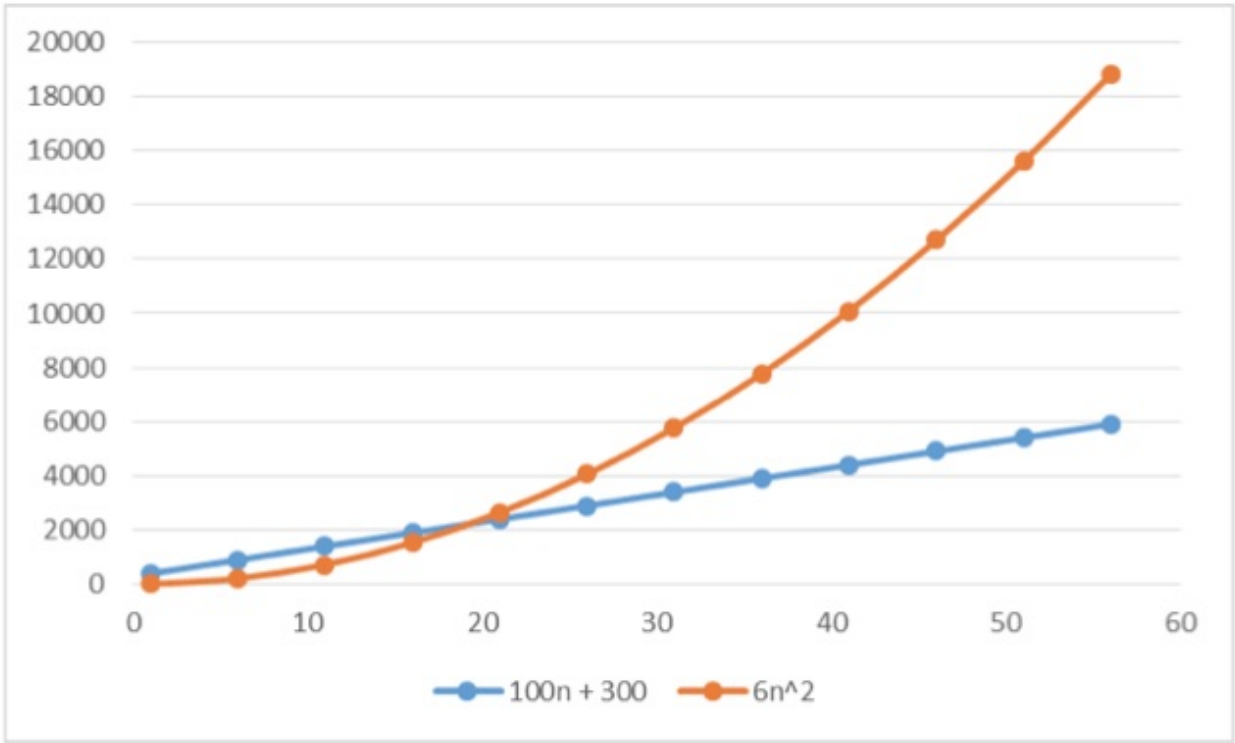
Complejidad algorítmica

¿Qué función crece más rápido?



Complejidad algorítmica

¿Qué función crece más rápido **asintóticamente**?



A partir de un $n_0, \forall n \geq n_0, T_2(n) \geq T_1(n)$

Complejidad algorítmica

1. La notación asintótica (tambien conocida como notacion "big-o"), o Big-O, es una notación matemática que permite caracterizar un algoritmo en términos de una función de referencia (logarítmica, lineal, cuadrática, polinomial, exponencial), en términos del tamaño de la entrada.
2. Indica el comportamiento límite de una función.

Complejidad algorítmica

1. La notación asintótica (tambien conocida como notacion "big-o"), o Big-O, es una notación matemática que permite caracterizar un algoritmo en términos de una función de referencia (logarítmica, lineal, cuadrática, polinomial, exponencial), en términos del tamaño de la entrada.
2. Indica el comportamiento límite de una función.

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones (**conjunto de funciones**) que difieren de $g(n)$ por una constante.

Complejidad algorítmica

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones que difieren de $g(n)$ por una constante.

La gráfica de esas funciones tienen la misma forma de $g(n)$, pero no son exactamente las mismas entre ellas. **EXHIBEN EL MISMO COMPORTAMIENTO, QUE LAS DIFERENCIA DE OTRAS FAMILIAS.**

Complejidad algorítmica

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones que difieren de $g(n)$ por una constante.

Ejemplo 1:

1. $T_1(n) = 1$
2. $T_3(n) = 10$
3. $T_4(n) = 200$

Todas ellas son funciones con el mismo comportamiento, solo difieren en una constante de la función $g(n) = 1$ (Función base).

Complejidad algorítmica

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones que difieren de $g(n)$ por una constante.

Ejemplo 1:

1. $T_1(n) = 1$
2. $T_3(n) = 10$
3. $T_4(n) = 200$

Todas ellas son funciones con el mismo comportamiento, solo difieren en una constante de la función $g(n) = 1$ (Función base).
T_1, T_2, T_3, T_4, **todas ellas están en** $O(1)$.

Complejidad algorítmica

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones que difieren de $g(n)$ por una constante.

Ejemplo 2:

1. $T_1(n) = 20n$
2. $T_2(n) = 2n$
3. $T_3(n) = 3n$

Todas ellas son funciones con el mismo comportamiento, que difieren en una constante de la función $g(n) = n$ (Función base).

Complejidad algorítmica

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones que difieren de $g(n)$ por una constante.

Ejemplo 2:

1. $T_1(n) = 20n$
2. $T_2(n) = 2n$
3. $T_3(n) = 3n$

Todas ellas son funciones con el mismo comportamiento, que difieren en una constante de la función $g(n) = n$ (Función base).

Todas ellas están en $O(n)$.

Complejidad algorítmica

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones que difieren de $g(n)$ por una constante.

Ejemplo 3:

1. El número de pasos para ejecutar `search(A,n,v)` es $T(n) = n$ en el peor caso. ¿$T(n) \in O(n)$?

Complejidad algorítmica

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones que difieren de $g(n)$ por una constante.

Ejemplo 3:

1. El número de pasos para ejecutar `search(A,n,v)` es $T(n) = n$ en el peor caso. ¿$T(n) \in O(n)$?
2. El número de pasos para ejecutar `maxArray(A,n)` es $T(n) = 6n - 1$ en el peor caso. ¿$T(n) \in O(n)$?

Complejidad algorítmica

Dada una función $g(n)$, denotamos por $O(g(n))$ a la familia de funciones que difieren de $g(n)$ por una constante.

Ejemplo 3:

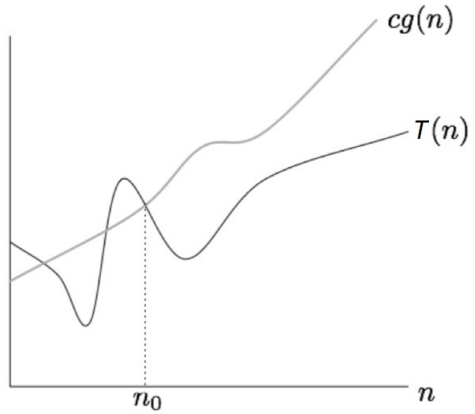
1. El número de pasos para ejecutar `search(A,n,v)` es $T(n) = n$ en el peor caso. ¿$T(n) \in O(n)$?
2. El número de pasos para ejecutar `maxArray(A,n)` es $T(n) = 6n - 1$ en el peor caso. ¿$T(n) \in O(n)$?

¿Tienen `search(A,n,v)` y `maxArray(A,n)` la misma complejidad?

Complejidad algorítmica

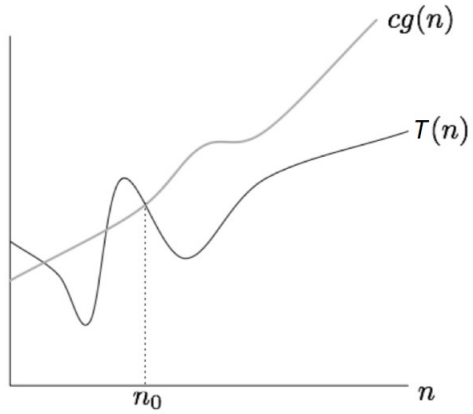
Dada $T(n)$ el número de operaciones básicas para ejecutar un algoritmo A, decimos que A tiene complejidad $g(n)$, o que $T(n)$ está en $O(g(n))$ si:

Existe una constante positiva c y un entero positivo n_0 tal que $T(n) \leq cg(n), \forall n_0 \leq n$.



Complejidad algorítmica

Existe una constante positiva c y un entero positivo n_0 tal que $T(n) \leq cg(n), \forall n_0 \leq n$.



Se lee:

1. "$T(n)$ es de orden $g(n)$"
2. $T(n)$ no crece más rápido que $g(n)$

Complejidad algorítmica

Receso de 10 minutos

Complejidad algorítmica

Suponga que después de diseñar el algoritmo A, se realiza su análisis, y resulta que el número de operaciones básicas que dicho algoritmo realiza es $T(n) = 2n + 10$.

¿Cuál es la complejidad algorítmica de A?, ¿Cómo la determinamos?.

Complejidad algorítmica

Suponga que después de diseñar el algoritmo A, se realiza su análisis, y resulta que el número de operaciones básicas que dicho algoritmo realiza es $T(n) = 2n + 10$.

¿Cuál es la complejidad algorítmica de A?, ¿Cómo la determinamos?.

R = **Encontrar** $g(n)$**, tal que** $T(n) \in O(g(n))$.

Complejidad algorítmica

Suponga que después de diseñar el algoritmo A, se realiza su análisis, y resulta que el número de operaciones básicas que dicho algoritmo realiza es $T(n) = 2n + 10$.

¿Cuál es la complejidad algorítmica de A?, ¿Cómo la determinamos?.

R = **Encontrar** $g(n)$**, tal que** $T(n) \in O(g(n))$.

Por el grado del polinomio, la hipótesis es que $T(n)$ es $O(n)$, es decir suponemos que $g(n) = n$. ¿Cómo lo demostramos?

Complejidad algorítmica

Suponga que después de diseñar el algoritmo A, se realiza su análisis, y resulta que el número de operaciones básicas que dicho algoritmo realiza es $T(n) = 2n + 10$.

¿Cuál es la complejidad algorítmica de A?, ¿Cómo la determinamos?.

R = **Encontrar** $g(n)$**, tal que** $T(n) \in O(g(n))$.

Por el grado del polinomio, la hipótesis es que $T(n)$ es $O(n)$, es decir suponemos que $g(n) = n$. ¿Cómo lo demostramos?

Debemos encontrar c **y** n_0 **tal que** $(2n + 10) \leq cn, \forall n \geq n_0$

15 minutos para encontrar la respuesta.

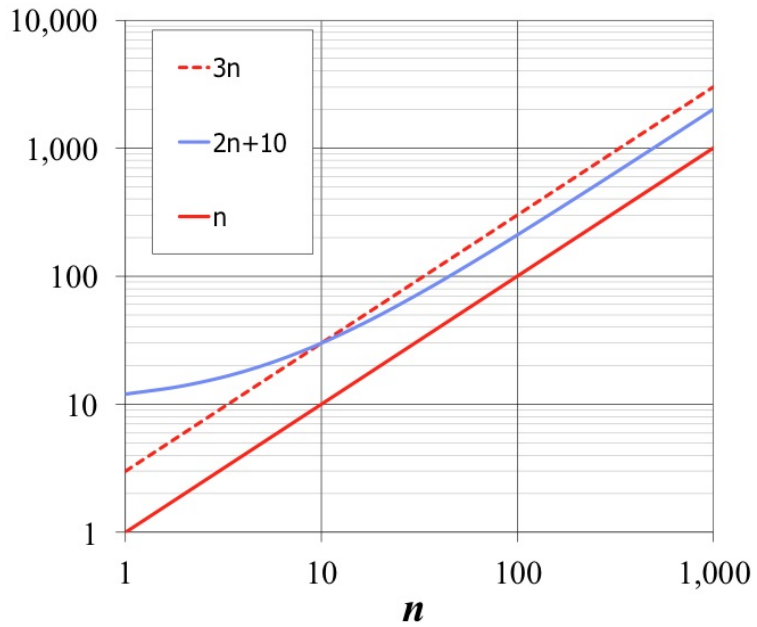
Complejidad algorítmica

$2n + 10 \leq cn$

$10 \leq n(c - 2)$

$10/(c - 2) \leq n$

Tomamos $c = 3,\ n_0 = 10$



Complejidad algorítmica

Sea $T(n) = n^2$. Demuestre que $T(n)$ NO es $O(n)$.

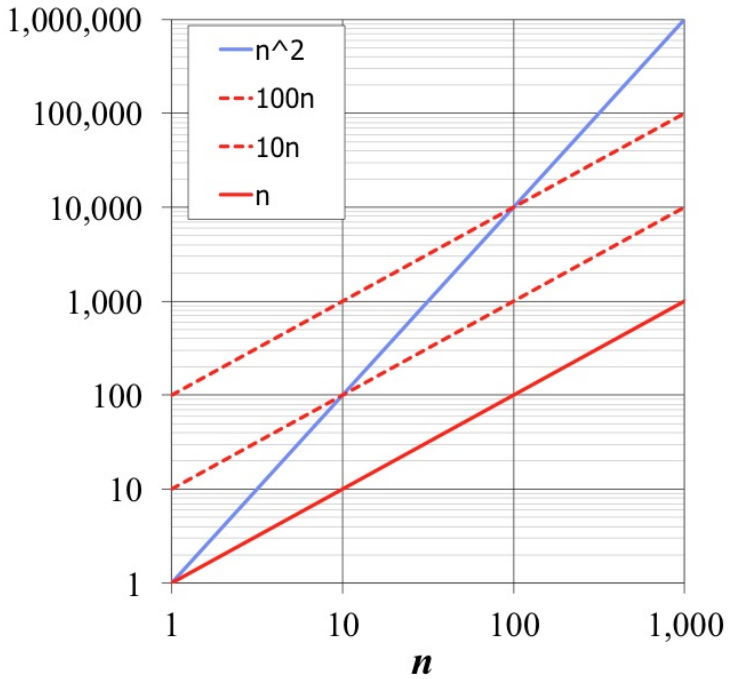
15 minutos para resolverlo.

Complejidad algorítmica

$n^2 \leq cn$

$n \leq c$

No existe una constante que cumpla con la condición anterior. Por lo tanto, no hay forma en la que n^2 no crezca más rápido que n.



Complejidad algorítmica

Sea $T(n) = 2n^3 + n + 1$

¿$T(n)$ está en $O(n^2)$?

Complejidad algorítmica

Sea $T(n) = 2n^3 + n + 1$

¿$T(n)$ está en $O(n^2)$?

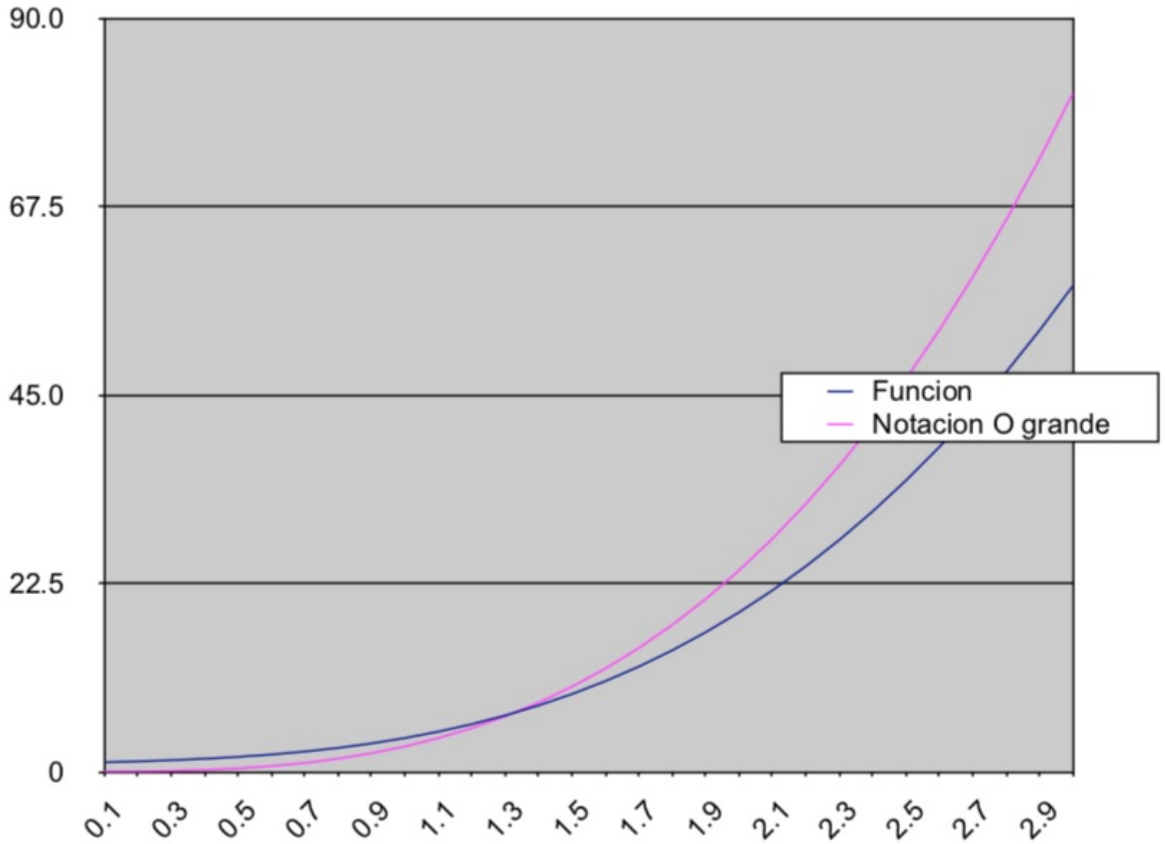
¿$T(n)$ está en $O(n^3)$?

Resuelva, determine c y n_0

Complejidad algorítmica

Sea $T(n) = 2n^3 + n + 1$

$T(n)$ es $O(n^3)$. Usa $c = 3$ y $n_0 = 2$.



Complejidad algorítmica

La notación "Big O" permite definir un límite superior al crecimiento de una función.

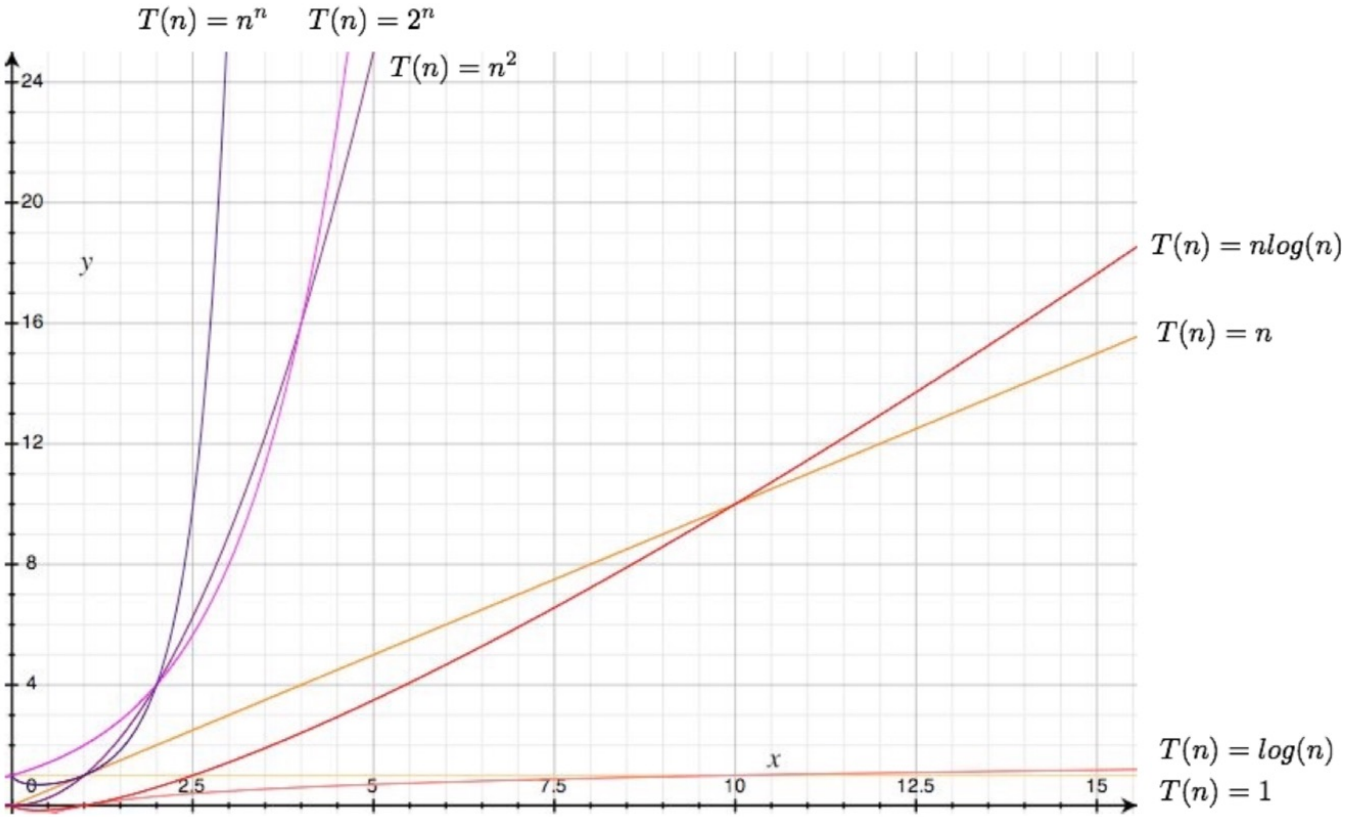
Establece un orden entre familias de funciones.

$$1 \leq \log n \leq n \leq n \log n \leq n^2 \leq n^3 \leq ... \leq n^p \leq a^n \leq n^n$$

Complejidad algorítmica

La notación "Big O" establece un orden entre familias de funciones.

$$1 \leq \log n \leq n \leq n \log n \leq n^2 \leq n^3 \leq ... \leq n^p \leq a^n \leq n^n$$



Complejidad algorítmica

La notación "Big O" establece un orden entre familias de funciones.
$1 \leq \log n \leq n \leq n \log n \leq n^2 \leq n^3 \leq ... \leq n^p \leq a^n \leq n^n$

Si $T(n)$ es $O(n)$, $T(n)$ también es $O(n^2)$ u $O(2^n)$, pero no es $O(1)$ u $O(\log n)$.

Complejidad algorítmica

La notación "Big O" establece un orden entre familias de funciones.
$1 \leq \log n \leq n \leq n \log n \leq n^2 \leq n^3 \leq ... \leq n^p \leq a^n \leq n^n$

Si $T(n)$ es $O(n)$, $T(n)$ también es $O(n^2)$ u $O(2^n)$, pero no es $O(1)$ u $O(\log n)$.

Cuando se determina la complejidad algoritmica, se debe determinar la familia de funciones más pequeña que satisface $T(n) \in O(g(n))$.

Complejidad algorítmica

Sea $T(n) = 2n^3 + n + 1$

¿Cuáles de las siguientes afirmaciones es correcta y cuál es falsa?

1. $T(n)$ es $O(2^n)$

Complejidad algorítmica

Sea $T(n) = 2n^3 + n + 1$

¿Cuáles de las siguientes afirmaciones es correcta y cuál es falsa?

1. $T(n)$ es $O(2^n)$
2. $T(n)$ es $O(n)$

Complejidad algorítmica

Sea $T(n) = 2n^3 + n + 1$

¿Cuáles de las siguientes afirmaciones es correcta y cuál es falsa?

1. $T(n)$ es $O(2^n)$
2. $T(n)$ es $O(n)$
3. $T(n)$ es $O(n^4)$

Complejidad algorítmica

Sea $T(n) = 2n^3 + n + 1$

¿Cuáles de las siguientes afirmaciones es correcta y cuál es falsa?

1. $T(n)$ es $O(2^n)$
2. $T(n)$ es $O(n)$
3. $T(n)$ es $O(n^4)$

¿Cual es la complejidad de $T(n)$?

Complejidad algorítmica

Sea $T(n) = 2n^3 + n + 1$

¿Cuáles de las siguientes afirmaciones es correcta y cuál es falsa?

1. $T(n)$ es $O(2^n)$
2. $T(n)$ es $O(n)$
3. $T(n)$ es $O(n^4)$

¿Cual es la complejidad de $T(n)$?

$T(n)$ es $O(n^3)$.

Se pueden **descartar** los términos de grado inferior y las constantes.

Complejidad algorítmica

Sea $T(n) = 10n^3 + 4n^2 + 1000$

¿Cuál es la complejidad de $T(n)$?

Complejidad algorítmica

Sea $T(n) = 10n^3 + 4n^2 + 1000$

¿Cuál es la complejidad de $T(n)$?

$T(n)$ es $O(n^3)$.

Complejidad algorítmica

Sea $T_1(n) = 10n^3 + 4n^2 + 1000$ el número de pasos del Algoritmo A para resolver P

Sea $T_2(n) = 2n^3 + n + 1$ el número de pasos del Algoritmo B que también resuelve P

¿Cuál algoritmo es mejor?

Complejidad algorítmica

Sea $T_1(n) = 10n^3 + 4n^2 + 1000$ el número de pasos del Algoritmo A para resolver P

Sea $T_2(n) = 2n^3 + n + 1$ el número de pasos del Algoritmo B que también resuelve P

¿Cuál algoritmo es mejor?

Ambos $T_1(n), T_2(n)$ son $O(n^3)$, tienen la misma complejidad.

Complejidad algorítmica

Sea $A(n)$ el algoritmo con la siguiente descripción.

$A(n)$\{
    $f_1(n)$
    $f_2(n)$
    $f_3(n)$
    for ($i = 1$ to n)
        $f_1(n)$
\}

Complejidad algorítmica

Sea $A(n)$ el algoritmo con la siguiente descripción.

$A(n)\{$

$f_1(n)$

$f_2(n)$

$f_3(n)$

for $(i = 1$ to $n)$

$f_1(n)$

$\}$

Sea $T(n)$ el número de pasos para ejecutar $A(n)$, $T(n)$ en $O(g(n))$. Suponga que la complejidad de $f_1(n)$ es $O(n^2)$, $f_2(n)$ es $O(1)$, $f_3(n)$ es $O(n)$.

Complejidad algorítmica

Sea $A(n)$ el algoritmo con la siguiente descripción.

$A(n)$\{
    $f_1(n)$
    $f_2(n)$
    $f_3(n)$
    for ($i = 1$ to n)
        $f_1(n)$
\}

Sea $T(n)$ el número de pasos para ejecutar $A(n)$, $T(n)$ en $O(g(n))$. Suponga que la complejidad de $f_1(n)$ es $O(n^2)$, $f_2(n)$ es $O(1)$, $f_3(n)$ es $O(n)$.

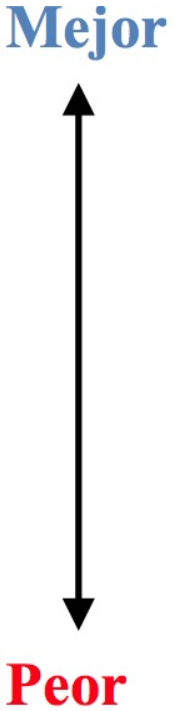
¿Quién es $g(n)$?

Complejidad algorítmica

Receso de 5 minutos

Complejidad algorítmica

Sea $T(n)$ la estimación de operaciones básicas del Algoritmo A:



- O(1) tiempo constante
- O(log n) tiempo logarítmico
- O(n) tiempo lineal
- O(n log n) tiempo lineal logarítmico
- $O(n^2)$ tiempo cuadrático
- $O(n^3)$ tiempo cúbico
- $O(n^k)$ tiempo polinomial
- $O(2^n)$ tiempo exponencial

Complejidad algorítmica

Expectativa del tiempo de ejecución de un algoritmo de acuerdo a su complejidad:

Complexity	10	20	50	100	1 000	10 000	100 000
O(1)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(log(n))	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n*log(n))	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
$O(n^2)$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	2 s	3-4 min
$O(n^3)$	< 1 s	< 1 s	< 1 s	< 1 s	20 s	5 hours	231 days
$O(2^n)$	< 1 s	< 1 s	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 s	hangs	hangs	hangs	hangs	hangs	hangs
$O(n^n)$	3-4 min	hangs	hangs	hangs	hangs	hangs	hangs

Ejercicios

¿Cuál es la complejidad del siguiente algoritmo?

```
1  int findMaxElement(int[] array) {
       int max = array[0];
3      for (int i = 0; i < array.length; i++) {
           if (array[i] > max) {
5              max = array[i];
           }
7      }
       return max;
9  }
```

Listing 1: Algoritmo 1

Ejercicios

¿Cuál es la complejidad del siguiente algoritmo?

```
1
   long findInversions(int[] array){
3        long inversions = 0;
         for (int i = 0; i < array.length; i++)
5            for (int j = i + 1; j < array.length; i++)
                 if (array[i] > array[j])
7                    inversions++;
       return inversions;
9  }
```

Listing 2: Algoritmo 2

Ejercicios

¿Cuál es la complejidad del siguiente algoritmo?

```
1
   int sum3(int n) {
3      decimal sum = 0;
       for (int a = 0; a < n; a++)
5          for (int b = 0; b < n; b++)
               for (int c = 0; c < n; c++)
7                  sum += a*b*c;
       return sum;
9  }
```

Listing 3: Algoritmo 3

Ejercicios

¿Cuál es la complejidad del siguiente algoritmo?

```
1
   int calculate(int n) {
3      int result = 0;
       for (int i = 0; i < (1<<n); i++)
5          result += i;
       return result;
7  }
```

Listing 4: Algoritmo 4

Ejercicios

¿Cuál es la complejidad del siguiente algoritmo?

```
1
   decimal Fibonacci(int n) {
3       if (n == 0)
            return 1;
5       else if (n == 1)
            return 1;
7       else
            return Fibonacci(n - 1) + Fibonacci(n - 2);
9  }
```

Listing 5: Algoritmo 5

Operaciones más comunes en programación sobre colecciones:

Colección	Acceso	Agregar	Encontrar	Eliminar
Array ($T[]$)	$O(1)$	$O(n)$	$O(n)$	$O(n)$
List (LinkedList<T>)	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Map (HashMap<T>)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
(TreeMap<T>)	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$

Complejidad en algoritmos de ordenamiento:

Algoritmo	Complejidad en tiempo	Complejidad en espacio
Quicksort	$O(n^2)$	$O(\log n)$
Mergesort	$O(n \log n)$	$O(n)$
Timsort	$O(n \log n)$	$O(n)$
Heapsort	$O(n \log n)$	$O(1)$
Bubble Sort	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(1)$
Tree Sort	$O(n^2)$	$O(n)$
Shell Sort	$O(n \log n^2)$	$O(1)$
Bucket Sort	$O(n^2)$	$O(n)$

FIN

DEL

CURSO