Estructuras de Datos

Introducción

Carlos Alberto Ramirez Restrepo

Programa de Ingeniería de Sistemas y Computación
Departamento de Electrónica y Ciencias de la Computación
Pontificia Universidad Javeriana
Cali, Colombia
carlosalbertoramirez@javerianacali.edu.co

Plan

1. Descripción Curso
2. Nociones Básicas
3. Complejidad Computacional
   - Cálculo de complejidad por conteo
   - Cálculo de complejidad por inspección
4. Análisis Asintótico

Plan

Información General

- ✓ Profesor: Carlos Alberto Ramírez Restrepo
- ✓ Código: 300CIP009
- ✓ Créditos: 3
- ✓ Horas de clase por semana: 5
- ✓ Salón de clase: Miércoles Palmas 3.2 y Viernes El Lago 3.1

Información General

- ✓ En este curso se estudiarán los fundamentos de los lenguajes de programación de alto nivel y se hará énfasis en tipos abstractos de datos y estructuras de datos.
- ✓ Se brindan las bases para la solución de problemas que pueden abordarse usando como herramientas un computador, lenguajes de alto nivel y estructuras de datos clásicas (listas, pilas, colas y tablas de direccionamiento).
- ✓ El componente práctico será intensivo a nivel de desarrollo de algoritmos y su análisis.

Objetivos

Al finalizar el curso los estudiantes podrán:

Identificar la eficiencia de un algoritmo

- ✓ Hallar la complejidad de algoritmos iterativos.
- ✓ Comparar diferentes algoritmos en términos de complejidad.
- ✓ Encontrar el tiempo de ejecución de un programa.
- ✓ Comparar tiempos de ejecución teóricos y prácticos.

Objetivos

Al finalizar el curso los estudiantes podrán:

Detallar las principales características de los lenguajes de programación de alto nivel

- ✓ Describir la estructura de un lenguaje de programación de alto nivel.
- ✓ Explicar la secuencia de compilación de un programa.
- ✓ Programar usando librerías y sus APIs.
- ✓ Identificar, explicar y corregir bugs en programas.
- ✓ Describir las características de las variables en lenguajes de programación de alto nivel (e.g. tipos de datos, identificadores, alcance, visibilidad).

Objetivos

Al finalizar el curso los estudiantes podrán:

Diseñar e implementar programas en el paradigma orientado a objetos

- ✓ Definir e implementar objetos y clases.
- ✓ Aplicar nociones de programación orientada a objetos.
- ✓ Usar principios de diseño orientado a objetos.
- ✓ Implementar programas en el paradigma orientado a objetos y realizar pruebas unitarias en ellos.

Objetivos

Diseñar e implementar soluciones a problemas computacionales mediante el uso de Tipos Abstractos de Datos (TADs) y Estructuras de datos

- ✓ Diseñar, implementar y evaluar tipos abstractos de datos.
- ✓ Diseñar, implementar y usar estructuras de datos lineales.
- ✓ Aplicar conceptos básicos de programación orientada a objetos en la implementación de estructuras de datos.
- ✓ Describir las características de las estructuras de datos en memoria secundaria.
- ✓ Utilizar e implementar los TADs básicos (listas, colas, pilas, árboles, grafos).

Metodología

- ✓ El curso se desarrollará mediante clases teórico/prácticas desarrolladas en el salón de clases.
- ✓ Eventualmente se propondrán tareas, talleres y ejercicios para entregar.
- ✓ Habrán tres parciales que evalúen los temas vistos y al final del curso deberá entregarse un proyecto.

Contenido

- ✓ Complejidad de algoritmos.
- ✓ Características de los lenguajes de programación y detalles de implementación.
- ✓ Lenguajes de alto nivel, compiladores y máquinas virtuales.
- ✓ Lenguaje C/C++, referencias, apuntadores, declaraciones y tipos.
- ✓ Depuración, tratamiento de excepciones.

Contenido

- ✓ Nociones fundamentales Programación orientada a objetos: objeto, clase, abstracción, herencia, polimorfismo.
- ✓ Definición de clases.
- ✓ Uso de clases, iteradores y otros componentes.
- ✓ Descomposición de programas, encapsulamiento y separación de comportamiento e implementación.

Contenido

- ✓ Tipos Abstractos de Datos (TAD).
- ✓ TAD lista, utilización, implementaciones (estructuras enlazadas, vectores, cursores).
- ✓ Teoría e implementación TAD Pila y TAD Cola.
- ✓ Colas de prioridad, conjuntos, *maps*.
- ✓ Teoría e implementación Tablas hash.
- ✓ Recorridos iterativos y recursivos sobre estructuras de datos.

Evaluación

- ✓ Tareas y talleres 20%
- ✓ Parcial 1: 20%
- ✓ Parcial 2: 20%
- ✓ Parcial 3: 20%
- ✓ Proyecto: 20%

Bibliografía

- ✓ *The C++ Programming Language (4th Edition)*. Bjarne Stroustrup. Addison-Wesley. May, 2013.
- ✓ *Programming: Principles and Practice Using C++ (2nd Edition)*. Bjarne Stroustrup. Addison-Wesley. May, 2014.
- ✓ *Data Structures and Algorithm Analysis in C++ (3rd Edition)*. Clifford A. Shaffer. Dover Publications. September, 2011.
- ✓ *Data Structures and Algorithms in C++ (2nd Edition)*. Michael T. Goodrich, Roberto Tamassia, and David M. Mount. Wiley. February, 2011.

Bibliografía

- ✓ *Data Structures with C++ Using STL (2nd Edition)*. William H. Ford and William R. Topp. Pearson. July, 2001.
- ✓ *Concepts of Programming Languages (11th Edition)*. Robert W. Sebesta. Pearson. February, 2015.
- ✓ *Diseño y manejo de estructuras de datos en C*. Jorge Villalobos. McGraw-Hill Interamericana. 1996.
- ✓ *Problem Solving with Algorithms and Data Structures using Python*. Bradley N. Miller and David L. Ranum. Franklin, Beedle & Associates Incorporated. 2006.

Plan

Nociones Básicas

Qué es un problema?

Nociones Básicas

Qué es un problema?

Un problema existe cuando el estado en el que se encuentra un sistema difiere del estado en que se desea que esté.

Nociones Básicas

Qué es un problema?

Un problema existe cuando el estado en el que se encuentra un sistema difiere del estado en que se desea que esté.

Cómo se puede dar solución a un problema?

Nociones Básicas

Qué es un problema?

Un problema existe cuando el estado en el que se encuentra un sistema difiere del estado en que se desea que esté.

Cómo se puede dar solución a un problema?

- ✓ La solución a un problema es una serie de pasos que llevan del estado en que están al estado que se desea.
- ✓ Existen muchos problemas en el mundo (más de los que el hombre puede resolver).
- ✓ Una gran cantidad de dichos problemas pueden resolverse usando el computador como herramienta.

Nociones Básicas

- ✓ El estudio de los problemas que se pueden resolver por medios computacionales tiene sus comienzos en la década de 1930.

Nociones Básicas

- ✓ El estudio de los problemas que se pueden resolver por medios computacionales tiene sus comienzos en la década de 1930.
- ✓ En ese entonces, D. Hilbert pretendía crear un sistema matemático formal, completo y consistente, en el que todos los problemas pudieran plantearse con precisión.

Nociones Básicas

- ✓ El estudio de los problemas que se pueden resolver por medios computacionales tiene sus comienzos en la década de 1930.
- ✓ En ese entonces, D. Hilbert pretendía crear un sistema matemático formal, completo y consistente, en el que todos los problemas pudieran plantearse con precisión.
- ✓ Además deseaba encontrar un algoritmo para determinar si una proposición, dentro del sistema, era verdadera o falsa.

Nociones Básicas

- ✓ El estudio de los problemas que se pueden resolver por medios computacionales tiene sus comienzos en la década de 1930.
- ✓ En ese entonces, D. Hilbert pretendía crear un sistema matemático formal, completo y consistente, en el que todos los problemas pudieran plantearse con precisión.
- ✓ Además deseaba encontrar un algoritmo para determinar si una proposición, dentro del sistema, era verdadera o falsa.
- ✓ Con este sistema cualquier problema bien definido se resolvería aplicando dicho algoritmo.

Nociones Básicas

- ✓ Después de varias investigaciones K. Godel demostró que el sistema planteado por Hilbert no se podía construir. Para ello publicó el famoso Teorema de Incompletitud.
- ✓ **Teorema de Incompletitud**: *Ningún sistema deductivo que contenga los teoremas de la aritmética, y con los axiomas recursivamente enumerables puede ser consistente y completo a la vez*.

Nociones Básicas

- ✓ Posteriormente, se demostró que existen problemas que son indecidibles, es decir, no hay algoritmos que resuelvan dichos problemas (A. Church y A. Turing probaron que el problema de Hilbert era indecidible).

Nociones Básicas

- ✓ Posteriormente, se demostró que existen problemas que son indecidibles, es decir, no hay algoritmos que resuelvan dichos problemas (A. Church y A. Turing probaron que el problema de Hilbert era indecidible).
- ✓ A partir de esto, los problemas se dividieron en dos tipos: tratables e intratables.

Nociones Básicas

- ✓ Posteriormente, se demostró que existen problemas que son indecidibles, es decir, no hay algoritmos que resuelvan dichos problemas (A. Church y A. Turing probaron que el problema de Hilbert era indecidible).
- ✓ A partir de esto, los problemas se dividieron en dos tipos: tratables e intratables.
- ✓ Los estudios teóricos de los problemas siguieron cuando Church introdujo a las matemáticas una notación formal para las funciones calculables, que denominó cálculo lambda.
- ✓ La idea era transformar todas las fórmulas matemáticas a una forma estándar, de tal manera que la demostración de teoremas se convertiría en la transformación de cadenas de simbolos siguiendo un conjunto de reglas como en un sistema lógico.

Nociones Básicas

- ✓ Por otro lado, Turing argumentó que el problema de Hilbert podía ser atacado con la ayuda de una máquina.
- ✓ La Máquina de Turing podía ser usada por una persona para ejecutar un procedimiento bien definido, mediante el cambio del contenido de una cinta ilimitada, dividida en cuadros que pueden contener un solo símbolo de un conjunto dado (el alfabeto).

Plan

Generalidades

- ✓ Como se vio anteriormente, de forma general, un problema se denomina no decidible cuando no es posible encontrar un algoritmo que lo resuelva.

Generalidades

- ✓ Como se vio anteriormente, de forma general, un problema se denomina no decidible cuando no es posible encontrar un algoritmo que lo resuelva.
- ✓ Los problemas para los cuales existe un algoritmo que los resuelva se denominan decidibles.

Generalidades

- ✓ Como se vio anteriormente, de forma general, un problema se denomina no decidible cuando no es posible encontrar un algoritmo que lo resuelva.
- ✓ Los problemas para los cuales existe un algoritmo que los resuelva se denominan decidibles.
- ✓ Por otro lado, los problemas computacionales pueden ser también clasificados de acuerdo a su complejidad en problemas tratables y problemas intratables.

Generalidades

- ✓ Como se vio anteriormente, de forma general, un problema se denomina no decidible cuando no es posible encontrar un algoritmo que lo resuelva.
- ✓ Los problemas para los cuales existe un algoritmo que los resuelva se denominan decidibles.
- ✓ Por otro lado, los problemas computacionales pueden ser también clasificados de acuerdo a su complejidad en problemas tratables y problemas intratables.
- ✓ La complejidad de un algoritmo mide el grado u orden de crecimiento que tiene el tiempo de ejecución del algoritmo dado el tamaño de la entrada que tenga.

Generalidades

- ✓ Un problema es denominado intratable si, con entradas grandes, no puede ser resuelto por ningún computador, no importa lo rápido que sea, cuanta memoria tenga o cuanto tiempo se le de para que lo resuelva.

Generalidades

- ✓ Un problema es denominado intratable si, con entradas grandes, no puede ser resuelto por ningún computador, no importa lo rápido que sea, cuanta memoria tenga o cuanto tiempo se le de para que lo resuelva.
- ✓ Lo anterior sucede debido a que los algoritmos que existen para solucionar estos problemas tienen una complejidad muy grande.

Generalidades

Existen dos maneras de hallar la complejidad de un algoritmo:

1. por conteo, ó

Generalidades

Existen dos maneras de hallar la complejidad de un algoritmo:

1. por conteo, ó
2. por inspección o tanteo.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ Para encontrar la complejidad de un algoritmo por conteo se toma como referencia el número de veces que se realizan operaciones básicas.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ Para encontrar la complejidad de un algoritmo por conteo se toma como referencia el número de veces que se realizan operaciones básicas.
- ✓ Una operación básica corresponde a cualquier operación aritmética, asignación o acceso a memoria.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ Para encontrar la complejidad de un algoritmo por conteo se toma como referencia el número de veces que se realizan operaciones básicas.
- ✓ Una operación básica corresponde a cualquier operación aritmética, asignación o acceso a memoria.
- ✓ Por lo general, en cada linea de código se realiza una operación simple.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ Para encontrar la complejidad de un algoritmo por conteo se toma como referencia el número de veces que se realizan operaciones básicas.
- ✓ Una operación básica corresponde a cualquier operación aritmética, asignación o acceso a memoria.
- ✓ Por lo general, en cada linea de código se realiza una operación simple.
- ✓ Por simplicidad, se asume que todas las operaciones básicas cuestan igual.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ De esta manera, para calcular la complejidad de un algoritmo se debe tomar cada línea de código y determinar cuántas veces se ejecuta.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ De esta manera, para calcular la complejidad de un algoritmo se debe tomar cada línea de código y determinar cuántas veces se ejecuta.
- ✓ Luego, se debe sumar las cantidades encontradas y la complejidad será del orden del resultado dado.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ De esta manera, para calcular la complejidad de un algoritmo se debe tomar cada línea de código y determinar cuántas veces se ejecuta.
- ✓ Luego, se debe sumar las cantidades encontradas y la complejidad será del orden del resultado dado.
- ✓ Esta complejidad es una aproximación de cuánto se demorará todo el algoritmo en ejecutarse.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ La complejidad de un algoritmo se expresa como una función que mide la cantidad de operaciones básicas que se ejecutan.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ La complejidad de un algoritmo se expresa como una función que mide la cantidad de operaciones básicas que se ejecutan.
- ✓ Normalmente, el factor más importante que afecta el tiempo de ejecución de un algoritmo es el tamaño de la entrada.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ La complejidad de un algoritmo se expresa como una función que mide la cantidad de operaciones básicas que se ejecutan.
- ✓ Normalmente, el factor más importante que afecta el tiempo de ejecución de un algoritmo es el tamaño de la entrada.
- ✓ Para una entrada de tamaño n se expresa la complejidad o tiempo de ejecución T como una función de n y se escribe cómo $T(n)$.

Cálculo de complejidad

Cálculo de complejidad por conteo

- ✓ La complejidad de un algoritmo se expresa como una función que mide la cantidad de operaciones básicas que se ejecutan.
- ✓ Normalmente, el factor más importante que afecta el tiempo de ejecución de un algoritmo es el tamaño de la entrada.
- ✓ Para una entrada de tamaño n se expresa la complejidad o tiempo de ejecución T como una función de n y se escribe cómo $T(n)$.
- ✓ Claramente, $T(n)$ siempre es positivo.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

Considere el siguiente algoritmo:

```
void imprime100(){
   int i = 1;
   while(i <= 100){
     printf("%d",i);
     i++;
   }
}
```

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

Para calcular la complejidad de este algoritmo por conteo se puede utilizar una tabla donde se numeran las líneas de código y se determine el número de veces que se ejecuta cada una:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprime100(){`	
2	`  int i = 1;`	1
3	`  while(i <= 100){`	101
4	`  printf("%d",i);`	100
5	`  i++;`	100
6	`  }`	
7	`}`	

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

Para calcular la complejidad de este algoritmo por conteo se puede utilizar una tabla donde se numeran las líneas de código y se determine el número de veces que se ejecuta cada una:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprime100(){`	
2	`  int i = 1;`	1
3	`  while(i <= 100){`	101
4	`  printf("%d",i);`	100
5	`  i++;`	100
6	`  }`	
7	`}`	

La suma de las cantidades encontradas es:

$$\text{Número total ejecuciones} = 1 + 101 + 100 + 100 = 302$$

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

Para calcular la complejidad de este algoritmo por conteo se puede utilizar una tabla donde se numeran las líneas de código y se determine el número de veces que se ejecuta cada una:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprime100(){`	
2	`  int i = 1;`	1
3	`  while(i <= 100){`	101
4	`  printf("%d",i);`	100
5	`  i++;`	100
6	`  }`	
7	`}`	

La suma de las cantidades encontradas es:

$$\text{Número total ejecuciones} = 1 + 101 + 100 + 100 = 302$$

En consecuencia la complejidad del algoritmo es $T(n) = 302$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

- ✓ Dado que $T(n) = 302$ es una función constante, luego se tiene que la complejidad del algoritmo anterior es $O(1)$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

- ✓ Dado que $T(n) = 302$ es una función constante, luego se tiene que la complejidad del algoritmo anterior es $O(1)$.
- ✓ $T(n) = O(1)$ es la notación utilizada para expresar que el tiempo de computo de un algoritmo es una función constante con respecto al tamaño de la entrada.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 2

Considere el siguiente algoritmo:

```
void imprimeN(int n){
   int i = 1;
   while(i <= n){
     printf("%d",i);
     i++;
   }
}
```

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

Se enumeran las líneas y se procede a contabilizar:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprimeN(int n){`	
2	`  int i = 1;`	1
3	`  while(i <= n){`	$n+1$
4	`  printf("%d",i);`	n
5	`  i++;`	n
6	`  }`	
7	`}`	

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

Se enumeran las líneas y se procede a contabilizar:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprimeN(int n){`	
2	`  int i = 1;`	1
3	`  while(i <= n){`	$n+1$
4	`  printf("%d",i);`	n
5	`  i++;`	n
6	`  }`	
7	`}`	

La suma de las cantidades encontradas es:

$$\text{Número total ejecuciones} = 1 + (n+1) + n + n = 3n + 2$$

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

Se enumeran las líneas y se procede a contabilizar:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprimeN(int n){`	
2	`  int i = 1;`	1
3	`  while(i <= n){`	$n+1$
4	`  printf("%d",i);`	n
5	`  i++;`	n
6	`  }`	
7	`}`	

La suma de las cantidades encontradas es:

$$\text{Número total ejecuciones} = 1 + (n+1) + n + n = 3n + 2$$

En consecuencia la complejidad del algoritmo es $T(n) = 3n + 2$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

- ✓ Dado que $T(n) = 3n + 2$ es una función lineal, luego se tiene que la complejidad del algoritmo anterior es $O(n)$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo

- ✓ Dado que $T(n) = 3n + 2$ es una función lineal, luego se tiene que la complejidad del algoritmo anterior es $O(n)$.
- ✓ $T(n) = O(n)$ es la notación utilizada para expresar que el tiempo de computo del algoritmo es una función lineal del tamaño de la entrada.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

Considere el siguiente algoritmo:

```
void imprime_mitad(int n){
  int i = n;
  while(i > 0){
    printf("%d",i);
    i = i / 2;
  }
}
```

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ A primera vista, el algoritmo anterior es similar a los algoritmos de los ejemplos anteriores.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ A primera vista, el algoritmo anterior es similar a los algoritmos de los ejemplos anteriores.
- ✓ No obstante, en cada iteración el valor de i es reducido a la mitad.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ A primera vista, el algoritmo anterior es similar a los algoritmos de los ejemplos anteriores.
- ✓ No obstante, en cada iteración el valor de i es reducido a la mitad.
- ✓ Entonces en lugar de imprimirse los números de 1 a n, en realidad se imprimirán los números n, $n/2$, $n/4$, $n/8$, $\ldots$, $n/2^k$ (división entera).

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ A primera vista, el algoritmo anterior es similar a los algoritmos de los ejemplos anteriores.
- ✓ No obstante, en cada iteración el valor de i es reducido a la mitad.
- ✓ Entonces en lugar de imprimirse los números de 1 a n, en realidad se imprimirán los números n, $n/2$, $n/4$, $n/8$, $\ldots$, $n/2^k$ (división entera).
- ✓ De esta manera, cuando i tome el valor de 0 el ciclo terminará.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ A primera vista, el algoritmo anterior es similar a los algoritmos de los ejemplos anteriores.
- ✓ No obstante, en cada iteración el valor de i es reducido a la mitad.
- ✓ Entonces en lugar de imprimirse los números de 1 a n, en realidad se imprimirán los números n, $n/2$, $n/4$, $n/8$, $\ldots$, $n/2^k$ (división entera).
- ✓ De esta manera, cuando i tome el valor de 0 el ciclo terminará.
- ✓ Es fácil notar que n es dividido por 2 elevado a la potencia 0, 1, 2, $\ldots$, k ($k+1$ veces).
- ✓ En consecuencia, la condición $i > 0$ debe ser evaluada $k+2$ veces, incluyendo cuando $i = 0$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ De lo anterior, es requerido encontrar la iteración en la cual el algoritmo finaliza, osea el punto donde $n = 2^k$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ De lo anterior, es requerido encontrar la iteración en la cual el algoritmo finaliza, osea el punto donde $n = 2^k$.
- ✓ Para hallar el valor de k, se utilizan las propiedades de los logaritmos:

$$n = 2^k$$
$$log_2\ n = log_2\ 2^k$$
$$log_2\ n = k$$

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ De lo anterior, es requerido encontrar la iteración en la cual el algoritmo finaliza, osea el punto donde $n = 2^k$.
- ✓ Para hallar el valor de k, se utilizan las propiedades de los logaritmos:

$$n = 2^k$$
$$log_2\ n = log_2\ 2^k$$
$$log_2\ n = k$$

- ✓ En consecuencia, el número de iteraciones que se realizan es $log_2\ n\ + 2$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

De esta manera se tiene lo siguiente:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprime_mitad(int n){`	
2	`int i = n;`	1
3	`while(i > 0){`	$log_2\ n + 2$
4	`printf("%d",i);`	$log_2\ n + 1$
5	`i = i/2;`	$log_2\ n + 1$
6	`}`	
7	`}`	

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

De esta manera se tiene lo siguiente:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprime_mitad(int n){`	
2	`int i = n;`	1
3	`while(i > 0){`	$log_2\ n + 2$
4	`printf("%d",i);`	$log_2\ n + 1$
5	`i = i/2;`	$log_2\ n + 1$
6	`}`	
7	`}`	

La suma de las cantidades encontradas es:

$$\begin{aligned}\text{Número total ejecuciones} &= 1 + (log_2\ n + 2) + (log_2\ n + 1) + (log_2\ n + 1) \\ &= 3 * log_2\ n + 5\end{aligned}$$

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

De esta manera se tiene lo siguiente:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprime_mitad(int n){`	
2	`  int i = n;`	1
3	`  while(i > 0){`	$log_2\ n + 2$
4	`  printf("%d",i);`	$log_2\ n + 1$
5	`  i = i/2;`	$log_2\ n + 1$
6	`  }`	
7	`}`	

La suma de las cantidades encontradas es:

$$\begin{aligned}\text{Número total ejecuciones} &= 1 + (log_2\ n + 2) + (log_2\ n + 1) + (log_2\ n + 1) \\ &= 3 * log_2\ n + 5\end{aligned}$$

En consecuencia la complejidad del algoritmo es $T(n) = 3 * log_2\ n + 5$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ Dado que $T(n) = 3 * log_2\ n + 5$ es una función logarítmica, luego se tiene que la complejidad del algoritmo anterior es $O(log_2\ n)$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 3

- ✓ Dado que $T(n) = 3 * log_2\ n + 5$ es una función logarítmica, luego se tiene que la complejidad del algoritmo anterior es $O(log_2\ n)$.
- ✓ $T(n) = O(log_2\ n)$ es la notación utilizada para expresar que el tiempo de computo del algoritmo es una función logarítmica del tamaño de la entrada.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 4

Considere el siguiente algoritmo:

```
void imprimeNxN(int n){
   for(int i=1; i<=n; i++){
      for(int j=1; j<=n; j++){
        printf("%d",(i-1)*n + j);
      }
   }
}
```

Qué hace este algoritmo?

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 4

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void imprimeNxN(int n){`	
2	`  for(int i=1; i<=n; i++){`	$n+1$
3	`  for(int j=1; j<=n; j++){`	$n*(n+1)$
4	`  printf("%d",(i-1)*n + j);`	$n*n$
5	`  }`	
6	`  }`	
7	`}`	

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 4

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void imprimeNxN(int n){`	
2	`  for(int i=1; i<=n; i++){`	$n+1$
3	`  for(int j=1; j<=n; j++){`	$n*(n+1)$
4	`  printf("%d",(i-1)*n + j);`	$n*n$
5	`  }`	
6	`  }`	
7	`}`	

La suma de las cantidades encontradas es:

$$\text{Número total ejecuciones} = (n+1) + n*(n+1) + n*n = 2n^2 + 2n + 1$$

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 4

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void imprimeNxN(int n){`	
2	`  for(int i=1; i<=n; i++){`	$n+1$
3	`  for(int j=1; j<=n; j++){`	$n*(n+1)$
4	`  printf("%d",(i-1)*n + j);`	$n*n$
5	`  }`	
6	`  }`	
7	`}`	

La suma de las cantidades encontradas es:

$$\text{Número total ejecuciones} = (n+1) + n*(n+1) + n*n = 2n^2 + 2n + 1$$

En consecuencia la complejidad del algoritmo es $T(n) = 2n^2 + 2n + 1$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 4

- ✓ Dado que $T(n) = 2n^2 + 2n + 1$ es una función cuadrática, luego se tiene que la complejidad del algoritmo anterior es $O(n^2)$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 4

- ✓ Dado que $T(n) = 2n^2 + 2n + 1$ es una función cuadrática, luego se tiene que la complejidad del algoritmo anterior es $O(n^2)$.
- ✓ $T(n) = O(n^2)$ es la notación utilizada para expresar que el tiempo de computo del algoritmo es una función cuadrática del tamaño de la entrada.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

Considere el siguiente algoritmo:

```
int sumaVector(int *v, int n){
   int i = 0;
   int sum = 0;
   while(i < n){
     if(v[i] % 2 != 0)
       sum = sum + v[i];
     i++;
   }
   return sum;
}
```

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

Considere el siguiente algoritmo:

```
int sumaVector(int *v, int n){
   int i = 0;
   int sum = 0;
   while(i < n){
     if(v[i] % 2 != 0)
       sum = sum + v[i];
     i++;
   }
   return sum;
}
```

Qué cálculo realiza este algoritmo?

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ Cuando se analizan algoritmos con condicionales (como el algoritmo anterior) hay que tener en cuenta que el conteo se hace considerando si las guardas de los condicionales se cumplen o no.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ Cuando se analizan algoritmos con condicionales (como el algoritmo anterior) hay que tener en cuenta que el conteo se hace considerando si las guardas de los condicionales se cumplen o no.
- ✓ La complejidad en estos algoritmos se halla:

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ Cuando se analizan algoritmos con condicionales (como el algoritmo anterior) hay que tener en cuenta que el conteo se hace considerando si las guardas de los condicionales se cumplen o no.
- ✓ La complejidad en estos algoritmos se halla:
  - en el peor de los casos (cuando se asume que las guardas de los condicionales siempre se cumplen),

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ Cuando se analizan algoritmos con condicionales (como el algoritmo anterior) hay que tener en cuenta que el conteo se hace considerando si las guardas de los condicionales se cumplen o no.
- ✓ La complejidad en estos algoritmos se halla:
  - en el peor de los casos (cuando se asume que las guardas de los condicionales siempre se cumplen),
  - el caso promedio (cuando se asume que las guardas algunas veces se cumplen y otras veces no)

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ Cuando se analizan algoritmos con condicionales (como el algoritmo anterior) hay que tener en cuenta que el conteo se hace considerando si las guardas de los condicionales se cumplen o no.
- ✓ La complejidad en estos algoritmos se halla:
  - en el peor de los casos (cuando se asume que las guardas de los condicionales siempre se cumplen),
  - el caso promedio (cuando se asume que las guardas algunas veces se cumplen y otras veces no)y
  - el mejor de los casos (cuando se asume que las guardas no se cumplen).

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`int sumaVector(int *v, int n){`	
2	`  int i = 0;`	1
3	`  int sum = 0;`	1
4	`  while(i < n){`	$n+1$
5	`  if(v[i] % 2 != 0)`	n
6	`  sum = sum + v[i];`	?
7	`  i++;`	n
8	`  }`	
9	`  return sum;`	1
10	`}`	

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ La cantidad de veces que se ejecuta la línea 6 es indefinida debido a que depende de si la guarda del condicional es verdadera o falsa.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ La cantidad de veces que se ejecuta la línea 6 es indefinida debido a que depende de si la guarda del condicional es verdadera o falsa.
- ✓ Por esta razón tenemos que analizar esta situación con respecto a los tres casos:

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ La cantidad de veces que se ejecuta la línea 6 es indefinida debido a que depende de si la guarda del condicional es verdadera o falsa.
- ✓ Por esta razón tenemos que analizar esta situación con respecto a los tres casos:
  - En el mejor de los casos ningún elemento del vector es impar por lo que la línea 6 no se ejecutaría nunca.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ La cantidad de veces que se ejecuta la línea 6 es indefinida debido a que depende de si la guarda del condicional es verdadera o falsa.
- ✓ Por esta razón tenemos que analizar esta situación con respecto a los tres casos:
  - En el mejor de los casos ningún elemento del vector es impar por lo que la línea 6 no se ejecutaría nunca.
  - En el caso promedio, aproximadamente la mitad de los elementos será impar y la otra mitad par. En este caso la línea se ejecutaría $n/2$ veces.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

- ✓ La cantidad de veces que se ejecuta la línea 6 es indefinida debido a que depende de si la guarda del condicional es verdadera o falsa.
- ✓ Por esta razón tenemos que analizar esta situación con respecto a los tres casos:
  - En el mejor de los casos ningún elemento del vector es impar por lo que la línea 6 no se ejecutaría nunca.
  - En el caso promedio, aproximadamente la mitad de los elementos será impar y la otra mitad par. En este caso la línea se ejecutaría $n/2$ veces.
  - En el peor caso todos los elementos del vector son impares por lo que siempre que se ejecute la línea 5 se ejecutará la línea 6. Luego esta línea se ejecutará n veces.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

La suma de las cantidades encontradas es entonces:

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

La suma de las cantidades encontradas es entonces:

- ✓ En el mejor de los casos:

  Número total ejecuciones $= 1+1+(n+1)+n+\mathbf{0}+n+1 = 3n+4$

- ✓ En el caso promedio:

  Número total ejecuciones $= 1+1+(n+1)+n+\mathbf{n/2}+n+1 = 7/2n+4$

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 5

La suma de las cantidades encontradas es entonces:

- ✓ En el mejor de los casos:

  Número total ejecuciones $= 1+1+(n+1)+n+\mathbf{0}+n+1 = 3n+4$

- ✓ En el caso promedio:

  Número total ejecuciones $= 1+1+(n+1)+n+\mathbf{n/2}+n+1 = 7/2n+4$

- ✓ En el peor de los casos:

  Número total ejecuciones $= 1+1+(n+1)+n+\mathbf{n}+n+1 = 4n+4$

Por lo tanto, la complejidad del algoritmo es, en este algoritmo particular, $T(n) = O(n)$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Considere ahora el siguiente algoritmo más complejo:

```
void algoritmo(int *a, int n){
   for(int j=1; j<n; j++){
      int clave = a[j];
      int i = j-1:
      while(i>=0 && a[i] > clave){
        a[i+1] = a[i];
        i = i-1;
      }
      a[i+1] = clave;
  }
}
```

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Considere ahora el siguiente algoritmo más complejo:

```
void algoritmo(int *a, int n){
   for(int j=1; j<n; j++){
      int clave = a[j];
      int i = j-1:
      while(i>=0 && a[i] > clave){
        a[i+1] = a[i];
        i = i-1;
      }
      a[i+1] = clave;
  }
}
```

Qué cálculo realiza este algoritmo?

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void algoritmo(int *a, int n){`	
2	`  for(int j=1; j<n; j++){`	n
3	`  int clave = a[j];`	$n-1$
4	`  int i = j-1:`	$n-1$
5	`  while(i>=0 && a[i] > clave){`	$\mathbf{t_1 + t_2 + \ldots + t_{n-1}}$
6	`  a[i+1] = a[i];`	$\mathbf{(t_1 - 1) + \ldots + (t_{n-1} - 1)}$
7	`  i = i-1;`	$\mathbf{(t_1 - 1) + \ldots + (t_{n-1} - 1)}$
8	`  }`	
9	`  a[i+1] = clave;`	$n-1$
10	`  }`	
11	`}`	

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void algoritmo(int *a, int n){`	
2	`  for(int j=1; j<n; j++){`	n
3	`  int clave = a[j];`	$n-1$
4	`  int i = j-1:`	$n-1$
5	`  while(i>=0 && a[i] > clave){`	$\sum_{j=1}^{n-1} t_j$
6	`  a[i+1] = a[i];`	$\sum_{j=1}^{n-1} (t_j - 1)$
7	`  i = i-1;`	$\sum_{j=1}^{n-1} (t_j - 1)$
8	`  }`	
9	`  a[i+1] = clave;`	$n-1$
10	`  }`	
11	`}`	

Donde cada t_j corresponde al número de veces que se cumple la condición del **while** en la iteración j.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void algoritmo(int *a, int n){`	
2	`  for(int j=1; j<n; j++){`	n
3	`  int clave = a[j];`	$n-1$
4	`  int i = j-1:`	$n-1$
5	`  while(i>=0 && a[i] > clave){`	$\sum_{j=1}^{n-1} t_j$
6	`  a[i+1] = a[i];`	$\sum_{j=1}^{n-1} (t_j - 1)$
7	`  i = i-1;`	$\sum_{j=1}^{n-1} (t_j - 1)$
8	`  }`	
9	`  a[i+1] = clave;`	$n-1$
10	`  }`	
11	`}`	

En el mejor caso cúanto vale cada t_j?

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void algoritmo(int *a, int n){`	
2	`  for(int j=1; j<n; j++){`	n
3	`  int clave = a[j];`	$n-1$
4	`  int i = j-1:`	$n-1$
5	`  while(i>=0 && a[i] > clave){`	$\sum_{j=1}^{n-1} t_j$
6	`  a[i+1] = a[i];`	$\sum_{j=1}^{n-1} (t_j - 1)$
7	`  i = i-1;`	$\sum_{j=1}^{n-1} (t_j - 1)$
8	`  }`	
9	`  a[i+1] = clave;`	$n-1$
10	`  }`	
11	`}`	

En el mejor caso, $t_j = 1$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void algoritmo(int *a, int n){`	
2	`  for(int j=1; j<n; j++){`	n
3	`  int clave = a[j];`	$n-1$
4	`  int i = j-1:`	$n-1$
5	`  while(i>=0 && a[i] > clave){`	$\sum_{j=1}^{n-1} t_j$
6	`  a[i+1] = a[i];`	$\sum_{j=1}^{n-1} (t_j - 1)$
7	`  i = i-1;`	$\sum_{j=1}^{n-1} (t_j - 1)$
8	`  }`	
9	`  a[i+1] = clave;`	$n-1$
10	`  }`	
11	`}`	

En el peor caso cúanto vale cada t_j?

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void algoritmo(int *a, int n){`	
2	`  for(int j=1; j<n; j++){`	n
3	`  int clave = a[j];`	$n-1$
4	`  int i = j-1:`	$n-1$
5	`  while(i>=0 && a[i] > clave){`	$\sum_{j=1}^{n-1} t_j$
6	`  a[i+1] = a[i];`	$\sum_{j=1}^{n-1} (t_j - 1)$
7	`  i = i-1;`	$\sum_{j=1}^{n-1} (t_j - 1)$
8	`  }`	
9	`  a[i+1] = clave;`	$n-1$
10	`  }`	
11	`}`	

En el peor caso, $t_j = j$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

Se enumeran las líneas y se procede a contabilizar:

# Línea	Línea de código	Número de ejecuciones
1	`void algoritmo(int *a, int n){`	
2	`for(int j=1; j<n; j++){`	n
3	`int clave = a[j];`	$n-1$
4	`int i = j-1:`	$n-1$
5	`while(i>=0 && a[i] > clave){`	$\sum_{\mathbf{j=1}}^{\mathbf{n-1}} \mathbf{t_j}$
6	`a[i+1] = a[i];`	$\sum_{\mathbf{j=1}}^{\mathbf{n-1}} (\mathbf{t_j - 1})$
7	`i = i-1;`	$\sum_{\mathbf{j=1}}^{\mathbf{n-1}} (\mathbf{t_j - 1})$
8	`}`	
9	`a[i+1] = clave;`	$n-1$
10	`}`	
11	`}`	

En el caso promedio, se supone que se necesitan $j/2$ comparaciones, esto es, $t_j = j/2$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

La suma de las cantidades encontradas es entonces:

- ✓ En el mejor caso ($t_j = 1$):

$$\begin{aligned} T(n) &= n + (n-1) + (n-1) + \sum_{j=1}^{n-1} \mathbf{t_j} + \sum_{j=1}^{n-1} \mathbf{t_j} - \mathbf{1} + \sum_{j=1}^{n-1} \mathbf{t_j} - \mathbf{1} + (n-1) \\ &= n + (n-1) + (n-1) + \sum_{j=1}^{n-1} \mathbf{1} + \sum_{j=1}^{n-1} \mathbf{0} + \sum_{j=1}^{n-1} \mathbf{0} + (n-1) \\ &= n + (n-1) + (n-1) + (n-1) + (n-1) = 5n - 4 \end{aligned}$$

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

La suma de las cantidades encontradas es entonces:

- ✓ En el mejor caso ($t_j = 1$):

$$\begin{aligned} T(n) &= n + (n-1) + (n-1) + \sum_{j=1}^{n-1} \mathbf{t_j} + \sum_{j=1}^{n-1} \mathbf{t_j} - \mathbf{1} + \sum_{j=1}^{n-1} \mathbf{t_j} - \mathbf{1} + (n-1) \\ &= n + (n-1) + (n-1) + \sum_{j=1}^{n-1} \mathbf{1} + \sum_{j=1}^{n-1} \mathbf{0} + \sum_{j=1}^{n-1} \mathbf{0} + (n-1) \\ &= n + (n-1) + (n-1) + (n-1) + (n-1) = 5n - 4 \end{aligned}$$

Por lo tanto, la complejidad del algoritmo es en el mejor caso $T(n) = O(n)$.

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

La suma de las cantidades encontradas es entonces:

- ✓ En el peor caso ($t_j = j$):

$$T(n) = n + (n-1) + (n-1) + \sum_{j=1}^{n-1} \mathbf{t_j} + \sum_{j=1}^{n-1} \mathbf{t_j} - \mathbf{1} + \sum_{j=1}^{n-1} \mathbf{t_j} - \mathbf{1} + (n-1)$$

$$T(n) = n + (n-1) + (n-1) + \sum_{j=1}^{n-1} \mathbf{j} + \sum_{j=1}^{n-1} \mathbf{j} - \mathbf{1} + \sum_{j=1}^{n-1} \mathbf{j} - \mathbf{1} + (n-1)$$

$$= n + 3(n-1) + \frac{(n-1)*n}{2} + 2*\left(\frac{(n-1)*n}{2} - (n-1)\right)$$

$$= 4n - 3 + \left(\frac{(n-1)*n + 2n*(n-1) - n + 1}{2}\right)$$

$$= 4n - 3 + \left(\frac{3n^2 - 4n + 1}{2}\right) = \frac{3n}{2} + 2n - \frac{5}{2}$$

Cálculo de complejidad

Cálculo de complejidad por conteo - Ejemplo 6

La suma de las cantidades encontradas es entonces:

- ✓ En el peor caso ($t_j = j$):

$$T(n) = n + (n-1) + (n-1) + \sum_{j=1}^{n-1} \mathbf{t_j} + \sum_{j=1}^{n-1} \mathbf{t_j} - \mathbf{1} + \sum_{j=1}^{n-1} \mathbf{t_j} - \mathbf{1} + (n-1)$$

$$T(n) = n + (n-1) + (n-1) + \sum_{j=1}^{n-1} \mathbf{j} + \sum_{j=1}^{n-1} \mathbf{j} - \mathbf{1} + \sum_{j=1}^{n-1} \mathbf{j} - \mathbf{1} + (n-1)$$

$$= n + 3(n-1) + \frac{(n-1)*n}{2} + 2*\left(\frac{(n-1)*n}{2} - (n-1)\right)$$

$$= 4n - 3 + \left(\frac{(n-1)*n + 2n*(n-1) - n + 1}{2}\right)$$

$$= 4n - 3 + \left(\frac{3n^2 - 4n + 1}{2}\right) = \frac{3n}{2} + 2n - \frac{5}{2}$$

Por lo tanto, el algoritmo en el peor caso es $T(n) = O(n^2)$.

Cálculo de complejidad

Cálculo de complejidad por inspección

- ✓ Hallar la complejidad por inspección o tanteo, es un método más rápido pero impreciso y, si no se cuenta con la suficiente experiencia, poco confiable.

Cálculo de complejidad

Cálculo de complejidad por inspección

- ✓ Hallar la complejidad por inspección o tanteo, es un método más rápido pero impreciso y, si no se cuenta con la suficiente experiencia, poco confiable.
- ✓ Simplemente se mira la estructura del algoritmo y se siguen las tres siguientes reglas:

Cálculo de complejidad

Cálculo de complejidad por inspección

- ✓ Hallar la complejidad por inspección o tanteo, es un método más rápido pero impreciso y, si no se cuenta con la suficiente experiencia, poco confiable.
- ✓ Simplemente se mira la estructura del algoritmo y se siguen las tres siguientes reglas:

1. La complejidad de una asignación es $O(1)$.

Cálculo de complejidad

Cálculo de complejidad por inspección

- ✓ Hallar la complejidad por inspección o tanteo, es un método más rápido pero impreciso y, si no se cuenta con la suficiente experiencia, poco confiable.
- ✓ Simplemente se mira la estructura del algoritmo y se siguen las tres siguientes reglas:

1. La complejidad de una asignación es $O(1)$.
2. La complejidad de un condicional es 1 más el máximo entre la complejidad del cuerpo del condicional cuando la guarda es positiva y el cuerpo del condicional cuando la guarda es negativa.

Cálculo de complejidad

Cálculo de complejidad por inspección

- ✓ Hallar la complejidad por inspección o tanteo, es un método más rápido pero impreciso y, si no se cuenta con la suficiente experiencia, poco confiable.
- ✓ Simplemente se mira la estructura del algoritmo y se siguen las tres siguientes reglas:

1. La complejidad de una asignación es $O(1)$.
2. La complejidad de un condicional es 1 más el máximo entre la complejidad del cuerpo del condicional cuando la guarda es positiva y el cuerpo del condicional cuando la guarda es negativa.
3. La complejidad de un ciclo es el número de veces que se ejecuta el ciclo multiplicado por la complejidad del cuerpo del ciclo.

Plan

Análisis Asintótico

Complejidades

A continuación se resumen las diferentes complejidades que puede tener un algoritmo:

Complejidad	**Nombre**	**Complejidad**	**Nombre**
$O(1)$	Constante	$O(n^c)$, $c > 3$	Polinomial
$O(log\ n)$	Logarítmica	$O(2^n)$	
$O(n)$	Lineal	$O(3^n)$	
$O(nlog\ n)$		$O(c^n)$, $c > 3$	Exponencial
$O(n^2)$	Cuadrática	$O(n!)$	Factorial
$O(n^3)$	Cúbica		

Análisis Asintótico

Complejidades

A continuación se resumen las diferentes complejidades que puede tener un algoritmo:

Complejidad	**Nombre**	**Complejidad**	**Nombre**
$O(1)$	Constante	$O(n^c)$, $c > 3$	Polinomial
$O(log\ n)$	Logarítmica	$O(2^n)$	
$O(n)$	Lineal	$O(3^n)$	
$O(nlog\ n)$		$O(c^n)$, $c > 3$	Exponencial
$O(n^2)$	Cuadrática	$O(n!)$	Factorial
$O(n^3)$	Cúbica		

- ✓ Se dice que un problema es tratable si su complejidad es polinomial o menor.

Análisis Asintótico

Complejidades

A continuación se resumen las diferentes complejidades que puede tener un algoritmo:

Complejidad	**Nombre**	**Complejidad**	**Nombre**
$O(1)$	Constante	$O(n^c)$, $c > 3$	Polinomial
$O(log\ n)$	Logarítmica	$O(2^n)$	
$O(n)$	Lineal	$O(3^n)$	
$O(nlog\ n)$		$O(c^n)$, $c > 3$	Exponencial
$O(n^2)$	Cuadrática	$O(n!)$	Factorial
$O(n^3)$	Cúbica		

- ✓ Se dice que un problema es tratable si su complejidad es polinomial o menor.
- ✓ Además se tiene que:

$$n! \gg c^n \gg 3^n \gg 2^n \gg n^c \gg n^3 \gg n^2 \gg nlog\ n \gg n \gg log\ n \gg 1$$

Análisis Asintótico

Complejidades: Algoritmos Comunes

Complejidad	**Algoritmo**
$O(1)$	Sumar dos números
$O(log\ n)$	Búsqueda binaria, inserción en un montículo
$O(n)$	Búsqueda elemento máximo de una secuencia
$O(nlog\ n)$	Merge-sort, Heap-sort, Quick-sort
$O(n^2)$	Bubble-sort, Recorridos en matrices
$O(n^3)$	Multiplicación de matrices, Floyd-Warshall
$O(2^n)$	Búsqueda exhaustiva, subconjuntos
$n!$	Hallar todas las permutaciones

Análisis Asintótico

Complejidades

Si se asume por ejemplo que en una máquina una operación toma 10^{-9} segundos, entonces:

	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$
20	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	77 years
30	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1.07s$	
40	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	18.3 min	
50	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	13 days	
100	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	10^{13} years	
10^3	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1s$		
10^4	$< 0.01s$	$< 0.01s$	$< 0.01s$	$0.1s$	16.7 min		
10^5	$< 0.01s$	$< 0.01s$	$< 0.01s$	$10s$	11 days		
10^6	$< 0.01s$	$< 0.01s$	$0.02s$	16.7 min	31 years		
10^7	$< 0.01s$	$0.01s$	$0.23s$	1.16 days			
10^8	$< 0.01s$	$0.1s$	$2.66s$	115 days			
10^9	$< 0.01s$	$1s$	$29.9s$	31 years			

Análisis Asintótico

Análisis Asintótico

- ✓ El análisis asintótico se refiere al estudio del comportamiento de un algoritmo cuando el tamaño de la entrada crece o tiende a infinito (en el mismo sentido de los límites en cálculo).

Análisis Asintótico

Análisis Asintótico

- ✓ El análisis asintótico se refiere al estudio del comportamiento de un algoritmo cuando el tamaño de la entrada crece o tiende a infinito (en el mismo sentido de los límites en cálculo).
- ✓ En este tipo de análisis, se omiten todos los factores constantes en las complejidades ya que cuando la entrada se hace cada vez más grande dichos valores no son muy relevantes.

Análisis Asintótico

Análisis Asintótico

- ✓ El análisis asintótico se refiere al estudio del comportamiento de un algoritmo cuando el tamaño de la entrada crece o tiende a infinito (en el mismo sentido de los límites en cálculo).
- ✓ En este tipo de análisis, se omiten todos los factores constantes en las complejidades ya que cuando la entrada se hace cada vez más grande dichos valores no son muy relevantes.
- ✓ De esta manera, el análisis asintótico proporciona un modelo simplificado del tiempo de ejecución de los algoritmos que permite compararlos y ayuda a entender mejor su comportamiento.

Análisis Asintótico

Análisis Asintótico

- ✓ Con el análisis asintótico se busca comparar la complejidad $T(n)$ de un algoritmo con una función $f(n)$ con la cual está acotada.

Análisis Asintótico

Análisis Asintótico

- ✓ Con el análisis asintótico se busca comparar la complejidad $T(n)$ de un algoritmo con una función $f(n)$ con la cual está acotada.
- ✓ De esta manera, se consideran tres posibilidades:

Análisis Asintótico

Análisis Asintótico

- ✓ Con el análisis asintótico se busca comparar la complejidad $T(n)$ de un algoritmo con una función $f(n)$ con la cual está acotada.
- ✓ De esta manera, se consideran tres posibilidades:
  - $f(n)$ es un límite superior, esto significa que, $c * f(n) \geq T(n)$ cuando n se hace muy grande.

Análisis Asintótico

Análisis Asintótico

- ✓ Con el análisis asintótico se busca comparar la complejidad $T(n)$ de un algoritmo con una función $f(n)$ con la cual está acotada.
- ✓ De esta manera, se consideran tres posibilidades:
  - $f(n)$ es un límite superior, esto significa que, $c * f(n) \geq T(n)$ cuando n se hace muy grande. En esta caso se dice que $T(n) = O(f(n))$.

Análisis Asintótico

Análisis Asintótico

- ✓ Con el análisis asintótico se busca comparar la complejidad $T(n)$ de un algoritmo con una función $f(n)$ con la cual está acotada.
- ✓ De esta manera, se consideran tres posibilidades:
  - $f(n)$ es un límite superior, esto significa que, $c * f(n) \geq T(n)$ cuando n se hace muy grande. En esta caso se dice que $T(n) = O(f(n))$.
  - $f(n)$ es un límite inferior, esto significa que, $c * f(n) \leq T(n)$ cuando n se hace muy grande.

Análisis Asintótico

Análisis Asintótico

- ✓ Con el análisis asintótico se busca comparar la complejidad $T(n)$ de un algoritmo con una función $f(n)$ con la cual está acotada.
- ✓ De esta manera, se consideran tres posibilidades:
  - $f(n)$ es un límite superior, esto significa que, $c * f(n) \geq T(n)$ cuando n se hace muy grande. En esta caso se dice que $T(n) = O(f(n))$.
  - $f(n)$ es un límite inferior, esto significa que, $c * f(n) \leq T(n)$ cuando n se hace muy grande. En esta caso se dice que $T(n) = \Omega(f(n))$.

Análisis Asintótico

Análisis Asintótico

- ✓ Con el análisis asintótico se busca comparar la complejidad $T(n)$ de un algoritmo con una función $f(n)$ con la cual está acotada.
- ✓ De esta manera, se consideran tres posibilidades:
  - $f(n)$ es un límite superior, esto significa que, $c * f(n) \geq T(n)$ cuando n se hace muy grande. En esta caso se dice que $T(n) = O(f(n))$.
  - $f(n)$ es un límite inferior, esto significa que, $c * f(n) \leq T(n)$ cuando n se hace muy grande. En esta caso se dice que $T(n) = \Omega(f(n))$.
  - $f(n)$ es un límite inferior, esto significa que, $c_1 * f(n) \leq T(n)$ y es también un límite superior, osea que $c_2 * f(n) \geq T(n)$ cuando n se hace muy grande. En esta caso se dice que $T(n) = \Theta(f(n))$.

Introducción

Ejercicio

Calcule la complejidad del siguiente algoritmo:

```
int programa(int n){
   int s = 0;
   int i = 1;
   while(i <= n){
     int t = 0;
     int j = 1;
     while(j <= i){
        t = t + 1;
        j = j + 1;
     }
     s = s + t;
     i = i + 1;
   }
   return s;
}
```

Introducción

Ejercicio

Calcule la complejidad del siguiente algoritmo:

```
Algoritmo(valores[1..n]){
   suma=0
   contador=0
   for(i=1; i<n; i++){
      for(j=i+1; j<=n; j++){
         if(valores[i] < valores[j]){
            for(h=j; h<=n; h++){
              suma += valores[i]
            }
         }
         else{
            contador++
            break;
         }}}
   return contador
}
```

Preguntas

?