Análisis de Algoritmos
Lectura: Básicos

Rodolfo Conde
rodolfo.conde@itam.mx

Instituto Tecnológico Autónomo de México
Maestría en Ciencias de la Computación

31 de agosto de 2020

Contenido

Básicos

En esta lectura, aprenderemos las características básicas del análisis de algoritmos y los tipos de análisis elementales.
Aprenderemos como probar que un algoritmo es correcto y a calcular la complejidad del tiempo de ejecución del peor caso y del caso promedio. Haremos estos calculos usando notación asintótica, lo cual nos permite hacer el análisis enfocandonos en el orden de crecimiento de las funciones.
Nuestra referencia principal es [Cor+09].

A continuación

¿Qué es el pseudocódigo?

El pseudocódigo es como un lenguaje de programación genérico, el cual es muy util para especificar algoritmos y procedimientos, de tal forma que la especificación es independiente de los detalles de un lenguaje de programación en particular.

¿Qué es el pseudocódigo?

Sin embargo, los elementos generales del pseudocódigo son muy parecidos a los que encontramos en varios lenguajes de programación conocidos.

Elementos del pseudocódigo

if $i \geq maxval$ **then**
    $i = 0$ // Un comentario
else
    if $i + k \leq maxval$ **then**
        $i = i + k$
    end if
end if

Elementos del pseudocódigo

```
function INSERTION-SORT(A)
    for j = 2 to A.length do
        lallave = A[j]
        i = j − 1
        while i > 0 and A[i] > lallave do    // Inserta A[j] en A[1 . . . j − 1]
            A[i + 1] = A[i]
            i = i − 1
        end while
        A[i + 1] = lallave
    end for
end function
```

Otros detalles del pseudocódigo

Sobre las estructuras de datos, objetos compuestos y otros detalles:

Otros detalles del pseudocódigo

Sobre las estructuras de datos, objetos compuestos y otros detalles:

- Usamos la notación $X.a$ para acceder al atributo a del objeto X.

Otros detalles del pseudocódigo

Sobre las estructuras de datos, objetos compuestos y otros detalles:

- Usamos la notación $X.a$ para acceder al atributo a del objeto X.
- El valor especial NIL tiene un uso totalmente análogo al del valor NULL (null) en varios lenguajes de programación.

Otros detalles del pseudocódigo

Sobre las estructuras de datos, objetos compuestos y otros detalles:

- Usamos la notación $X.a$ para acceder al atributo a del objeto X.
- El valor especial NIL tiene un uso totalmente análogo al del valor NULL (null) en varios lenguajes de programación.
- El paso de parámetros a las funciones o procedimientos que usamos es por valor.

Otros detalles del pseudocódigo

Sobre las estructuras de datos, objetos compuestos y otros detalles:

- Usamos la notación $X.a$ para acceder al atributo a del objeto X.
- El valor especial NIL tiene un uso totalmente análogo al del valor NULL (null) en varios lenguajes de programación.
- El paso de parámetros a las funciones o procedimientos que usamos es por valor.
- El enunciado **error** indica que ha ocurrido un error por una falla en los parámetros u otro motivo.

A continuación

El problema de ordenar elementos

El problema de ordenar un conjunto de n elementos es fundamental en las Ciencias de la Computación con diversos usos y aplicaciones. Formalmente, esta especificado así:

Definición 1

El Problema del ordenamiento de números esta dado por las siguientes condiciones:

Entrada: Una secuencia de n números $\langle a_1, a_2, \ldots, a_n \rangle$

Salida: Un reordenamiento (permutación) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ de la secuencia de entrada, tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

El problema de ordenar elementos

Existe muchas y diversas soluciones para este problema fundamental, con diferentes complejidades y tiempos de ejecución. Comenzaremos nuestro estudio de algoritmos con una de estas soluciones: Insertion Sort.

El problema de ordenar elementos

Existe muchas y diversas soluciones para este problema fundamental, con diferentes complejidades y tiempos de ejecución. Comenzaremos nuestro estudio de algoritmos con una de estas soluciones: Insertion Sort.

Insertion sort

Sobre este algoritmo:

Insertion sort

Sobre este algoritmo:

- Eficiente en un número pequeño de elementos.

Insertion sort

Sobre este algoritmo:

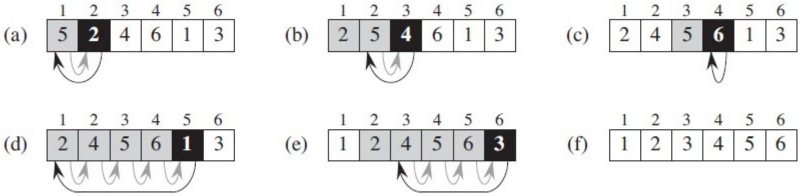
- Eficiente en un número pequeño de elementos.
- Funciona de la misma forma en que ordenamos un conjunto de cartas en la mano.

Insertion sort: Pseudocódigo

```
function INSERTION-SORT(A)
    for j = 1 to A.length − 1 do
        lallave = A[j]
        i = j − 1
        while i ≥ 0 and A[i] > lallave do    // Inserta A[j] en A[1 ... j − 1]
            A[i + 1] = A[i]
            i = i − 1
        end while
        A[i + 1] = lallave
    end for
end function
```

Insertion sort: Video demostrativo

Insertion sort: Ejemplo



Insertion sort

En la siguiente sección, comenzaremos el análisis de correctez de Insertion-Sort.

A continuación

Programas en pseudocódigo

Un problema clásico y elemental: Ordenar objetos

Análisis de correctez

Análisis de uso de recursos

Orden de crecimiento de funciones

Notas y ejercicios

Primeros pasos del análisis

El primer paso para ver que un algoritmo esta bien hecho, es verificar su correctez. Esto usualmente consta de dos pasos:

- Demostrar que tarda tiempo finito, de decir, que se detiene.
- Demostrar su correctez (que hace lo que se supone debe hacer).

Verificar que el tiempo es finito

Para verificar que el programa tarda tiempo finito (es decir, es un algoritmo), lo que usualmente hacemos es argumentar que los ciclos que el programa tiene son finitos, es decir, el bloque de cada ciclo se ejecuta un número finito de veces.

Verificar la correctez

Para verificar la correctez, es necesario (a grandes rasgos) analizar la ejecución del algoritmo en una entrada arbitraria. Hay que mostrar que durante la ejecución, se dan ciertas propiedades en la manipulación de las estructuras de datos involucradas y además, hay que ver que estas se conservan a lo largo de la ejecución. En particular, hay que verificar que ciertas propiedades se presentan y se conservan durante la ejecución de los ciclos que tenga el algoritmo. A estas propiedades las conocemos de manera formal como Invariantes del ciclo[1]

[1] Loop Invariants

Verificar la correctez

Hay tres propiedades que debemos mostrar que un Invariante del ciclo posee:

Inicialización El invariante es cierto antes de ejecutar la primera iteración.

Mantenimiento Si el invariante es cierto en una iteración anterior, es cierto en la siguiente.

Terminación El invariante es cierto al finalizar el ciclo y nos proporciona información útil que ayuda a mostrar la correctez del algoritmo.

Verificar la correctez

Hay tres propiedades que debemos mostrar que un Invariante del ciclo posee:

Inicialización El invariante es cierto antes de ejecutar la primera iteración.

Mantenimiento Si el invariante es cierto en una iteración anterior, es cierto en la siguiente.

Terminación El invariante es cierto al finalizar el ciclo y nos proporciona información útil que ayuda a mostrar la correctez del algoritmo.

¿Donde hemos visto esto antes?

Verificar la correctez

Inducción Matemática

$$3 + 7 + 11 \ldots (4n-1) = n(2n+1)$$

$$1^3 + 2^3 + 3 + \ldots n^3 = \frac{n^2(n+1)^2}{4}$$

$$1 + 2 + 2^2 + \ldots 2^{n-1} = 2^n - 1$$

Ejemplo: Insertion-Sort

Análisis de correctez de Insertion-Sort

Ejemplo: Insertion-Sort

Lema 2

El método Insertion-Sort es un algoritmo que para cualquier secuencia de entrada $\langle a_1, \ldots, a_n \rangle$, *produce una permutación* $\langle a'_1, \ldots, a'_n \rangle$ *tal que*

$$a'_1 \leq \ldots \leq a'_n. \tag{1}$$

Análisis de Insertion-Sort

Demostración.

Debemos mostrar que Insertion-Sort ejecuta un número finito de pasos en cualquier arreglo de entrada y que al finalizar, nos entrega el arreglo de entrada ordenado. Primero mostremos que tarda tiempo finito.

Análisis de Insertion-Sort: Correctez

```
1:  function INSERTION-SORT(A)
2:      for j = 1 to A.length − 1 do
3:          llave = A[j]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > llave do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = llave
10:     end for
11: end function
```

En efecto, Insertion-Sort contiene unicamente dos ciclos: El ciclo **for** de las lineas 2-10 y el ciclo **while** de las lineas 5-8.

Análisis de Insertion-Sort: Correctez

```
1:  function INSERTION-SORT(A)
2:      for j = 1 to A.length − 1 do
3:          llave = A[j]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > llave do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = llave
10:     end for
11: end function
```

El ciclo **for**, en la linea 2, realizará $n-1$ iteraciones, donde n es el tamaño del arreglo de entrada.

Análisis de Insertion-Sort: Correctez

```
1:  function INSERTION-SORT(A)
2:      for j = 1 to A.length − 1 do
3:          llave = A[j]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > llave do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = llave
10:     end for
11: end function
```

Y el ciclo **while** de la linea 5, realiza un número finito de iteraciones por el siguiente argumento:

Análisis de Insertion-Sort: Correctez

```
1: function Insertion-Sort(A)
2:     for j = 1 to A.length − 1 do
3:         llave = A[j]
4:         i = j − 1
5:         while i ≥ 0 and A[i] > llave do
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         end while
9:         A[i + 1] = llave
10:    end for
11: end function
```

Y el ciclo **while** de la linea 5, realiza un número finito de iteraciones por el siguiente argumento:

a) Si existe $i \in \{0, \ldots, j-1\}$ tal que $A[i] \leq llave$, entonces no hay nada que hacer.

Análisis de Insertion-Sort: Correctez

```
1:  function Insertion-Sort(A)
2:      for j = 1 to A.length − 1 do
3:          llave = A[j]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > llave do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = llave
10:     end for
11: end function
```

Y el ciclo **while** de la linea 5, realiza un número finito de iteraciones por el siguiente argumento:

b) Si $A[i] > llave$ para toda i, entonces el ciclo no puede tardar más de j iteraciones, pues el valor de i se disminuye en la linea 7.

Análisis de Insertion-Sort: Correctez

```
function INSERTION-SORT(A)
    for j = 1 to A.length − 1 do
        llave = A[j]
        i = j − 1
        while i ≥ 0 and A[i] > llave do
            A[i + 1] = A[i]
            i = i − 1
        end while
        A[i + 1] = llave
    end for
end function
```

Ahora vamos a probar que el algoritmo es correcto, es decir, que dada una secuencia de números de entrada $\langle a_1, \ldots, a_n \rangle$, el algoritmo permuta los elementos de la secuencia de tal forma que satisface la Ecuación (1).

Análisis de Insertion-Sort: Correctez

```
1:  function INSERTION-SORT(A)
2:      for j = 1 to A.length − 1 do
3:          llave = A[j]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > llave do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = llave
10:     end for
11: end function
```

Para probar la correctez de Insertion-Sort, basicamente necesitamos probar el siguiente invariante del ciclo para el **for** de la lineas 2-10:

Análisis de Insertion-Sort: Correctez

```
function INSERTION-SORT(A)
    for j = 1 to A.length − 1 do
        llave = A[j]
        i = j − 1
        while i ≥ 0 and A[i] > llave do
            A[i + 1] = A[i]
            i = i − 1
        end while
        A[i + 1] = llave
    end for
end function
```

Al iniciar la iteración j-ésima del ciclo **for**, el subarreglo $A[0 \ldots j-1]$ contiene elementos del arreglo original, tales que

$$A[0] \leq \cdots \leq A[j-1].$$

Análisis de Insertion-Sort: Correctez

```
1: function Insertion-Sort(A)
2:     for j = 1 to A.length − 1 do
3:         llave = A[j]
4:         i = j − 1
5:         while i ≥ 0 and A[i] > llave do
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         end while
9:         A[i + 1] = llave
10:    end for
11: end function
```

Inicialización. Al iniciar la ejecución del algoritmo, justo despues de la primera asignación de la linea 2 y antes de entrar en la primera iteración del **for** (linea 3), $j - 1 = 0$ y entonces el subarreglo $A[0 \ldots j-1]$ consiste del único elemento $A[0]$, el cual, esta ordenado. Por lo tanto, se cumple el invariante antes de iniciar el ciclo.

Análisis de Insertion-Sort: Correctez

```
1:  function INSERTION-SORT(A)
2:      for j = 1 to A.length − 1 do
3:          llave = A[j]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > llave do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = llave
10:     end for
11: end function
```

Mantenimiento. Supongamos que el invariante se satisface para la iteración j y vamos a probar que es cierto para la siguiente iteración. Al inicio, despues de la linea 3 la variable *llave* tiene el valor del elemento $A[j+1]$ del arreglo y por hipótesis, el subarreglo $A[0 \ldots j]$ esta ordenado

Análisis de Insertion-Sort: Correctez

```
1: function INSERTION-SORT(A)
2:     for j = 1 to A.length − 1 do
3:         llave = A[j]
4:         i = j − 1
5:         while i ≥ 0 and A[i] > llave do
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         end while
9:         A[i + 1] = llave
10:    end for
11: end function
```

y en el ciclo **while** de las lineas 5-8, el algoritmo mueve los elementos $A[0], A[j-1], \ldots$ y así sucesivamente hasta que se encuentra la posición adecuada para *llave* y este valor se guarda en esa posición en la linea 9. Por este hecho, el subarreglo $A[0 \ldots j+1]$ tiene todos sus elementos del arreglo original, pero ordenados.

Análisis de Insertion-Sort: Correctez

```
1:  function Insertion-Sort(A)
2:      for j = 1 to A.length − 1 do
3:          llave = A[j]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > llave do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = llave
10:     end for
11: end function
```

Por lo tanto, el invariante es cierto en cualquier iteración del ciclo **for** de las lineas 2-10.

Análisis de Insertion-Sort: Correctez

```
1:  function INSERTION-SORT(A)
2:      for j = 1 to A.length − 1 do
3:          llave = A[j]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > llave do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = llave
10:     end for
11: end function
```

Terminación. Para ver que el invariante nos ayuda a probar que el algoritmo es correcto y se satisface la Ecuación 1 para cualquier secuencia de números, Examinemos las variables al finalizar el ciclo **for**.

La condición que provóca que termine el ciclo **for** es cuando en la linea 2, $j > A.length - 1$, es decir, $j == n$, pues j se incrementa en 1 en cada iteración.

Por la condición de Mantenimiento, al finalizar el ciclo **for**, tenemos que el subarreglo $A[0 \ldots j-1]$ esta ordenado. Substituyendo el valor de j en esta expresión cuando termina el ciclo, tenemos que

$$A[0 \ldots n-1]$$

esta ordenado, pero esto es precisamente el arreglo de entrada y esto quiere decir que los elementos del arreglo satisfacen la Ecuación (1). Esto concluye la prueba del Lema. $\square$

Ejercicio

Considera el siguiente problema, llamado el Problema de busqueda:

Entrada: Una secuencia de n números $\langle a_1, \ldots, a_n \rangle$ y un valor v.

Salida: El primer indice i tal que $v == a_i$ o el valor NIL si v no esta presente en la secuencia

Escribe pseudocódigo para el algoritmo de busqueda lineal que recibe de entrada un arreglo A y el valor v y busca el primer elemento de A que es igual a v. Pruba que tu algoritmo es correcto.

A continuación

El uso de los recursos al ejecutar un algortimo

Despues de verificar la correctez de un algoritmo, podemos proceder a revisar la cantidad de recursos que utiliza. Nos interesan dos recursos fundamentales

El uso de los recursos al ejecutar un algortimo

Despues de verificar la correctez de un algoritmo, podemos proceder a revisar la cantidad de recursos que utiliza. Nos interesan dos recursos fundamentales

Tiempo computacional El tiempo que tarda el algoritmo en resolver una instancia del problema (CPU)

El uso de los recursos al ejecutar un algortimo

Despues de verificar la correctez de un algoritmo, podemos proceder a revisar la cantidad de recursos que utiliza. Nos interesan dos recursos fundamentales

Tiempo computacional
: El tiempo que tarda el algoritmo en resolver una instancia del problema (CPU)

Espacio
: Intuitivamente, la cantidad de bits que el algorito ocupa en resolver una instancia del problema (variables, arreglos, etc. que el algoritmo usa, adicional a la entrada)

El uso de los recursos al ejecutar un algortimo

Despues de verificar la correctez de un algoritmo, podemos proceder a revisar la cantidad de recursos que utiliza. Nos interesan dos recursos fundamentales

Tiempo computacional El tiempo que tarda el algoritmo en resolver una instancia del problema (CPU)

Espacio Intuitivamente, la cantidad de bits que el algorito ocupa en resolver una instancia del problema (variables, arreglos, etc. que el algoritmo usa, adicional a la entrada)

Comenzaremos analizando el tiempo computacional de los algoritmos.

Modelo de implementación

Antes de comenzar el analisis de tiempo, necesitamos establecer el modelo formal para realizar el analisis. En general, estaremos usando el modelo RAM[1]: Máquina de Acceso Alearorio para implementar nuestros algoritmos y hacer el analisis de recursos.

[1] Random Access Machine.

Modelo RAM

Las características fundamentales del modelo RAM son:

- Instrucciones simples (aritméticas, lógicas, nada complejo).
- Cada instrucción toma tiempo constante para su ejecución.
- Las instrucciones se ejecutan secuencialmente.
- Un único procesador.
- No existe la concurrencia (no procesos paralelos ni distribuidos).

¿Cómo comenzamos el analisis de tiempo de ejecución?

Tiempo de ejecución: Depende de la entrada

El tiempo de ejecución de un algoritmo dependera en general de la entrada. Esto ocurrira en dos formas fundamentales:

- El tamaño de la entrada.
- La forma y características propias de una entrada dada

Tiempo de ejecución: Depende de la entrada

El tiempo de ejecución de un algoritmo dependera en general de la entrada. Esto ocurrira en dos formas fundamentales:

- El tamaño de la entrada.
  - Entre más grande, mas tiempo tardará.
- La forma y características propias de una entrada dada

Tiempo de ejecución: Depende de la entrada

El tiempo de ejecución de un algoritmo dependera en general de la entrada. Esto ocurrira en dos formas fundamentales:

- El tamaño de la entrada.
  - Entre más grande, mas tiempo tardará.
- La forma y características propias de una entrada dada
  - Enradas de un mismo tamaño, pueden tomar tiempos muy diferentes.

Tiempo de ejecución: ¿Tamaño de la entrada?

La definición precisa de tamaño de entrada, dependerá en gran medida del tipo de problema que queremos resolver.

Tiempo de ejecución: ¿Tamaño de la entrada?

La definición precisa de tamaño de entrada, dependerá en gran medida del tipo de problema que queremos resolver.

- Ordenar objetos, Verificar la satisfacibilidad de una fórmula booleana, calcular la Transformada de Fourier Discreta

Tiempo de ejecución: ¿Tamaño de la entrada?

La definición precisa de tamaño de entrada, dependerá en gran medida del tipo de problema que queremos resolver.

- Ordenar objetos, Verificar la satisfacibilidad de una fórmula booleana, calcular la Transformada de Fourier Discreta $\rightarrow$ Número de objetos.

Tiempo de ejecución: ¿Tamaño de la entrada?

La definición precisa de tamaño de entrada, dependerá en gran medida del tipo de problema que queremos resolver.

- Ordenar objetos, Verificar la satisfacibilidad de una fórmula booleana, calcular la Transformada de Fourier Discreta $\rightarrow$ Número de objetos.
- Multiplicar enteros, verificar si un número es primo

Tiempo de ejecución: ¿Tamaño de la entrada?

La definición precisa de tamaño de entrada, dependerá en gran medida del tipo de problema que queremos resolver.

- Ordenar objetos, Verificar la satisfacibilidad de una fórmula booleana, calcular la Transformada de Fourier Discreta $\rightarrow$ Número de objetos.
- Multiplicar enteros, verificar si un número es primo $\rightarrow$ Número de bits para representar un número en binario.

Tiempo de ejecución: ¿Tamaño de la entrada?

La definición precisa de tamaño de entrada, dependerá en gran medida del tipo de problema que queremos resolver.

- Ordenar objetos, Verificar la satisfacibilidad de una fórmula booleana, calcular la Transformada de Fourier Discreta $\rightarrow$ Número de objetos.
- Multiplicar enteros, verificar si un número es primo $\rightarrow$ Número de bits para representar un número en binario.
- Verificar si $G = (V, E)$ es conexa, plana, etc.

Tiempo de ejecución: ¿Tamaño de la entrada?

La definición precisa de tamaño de entrada, dependerá en gran medida del tipo de problema que queremos resolver.

- Ordenar objetos, Verificar la satisfacibilidad de una fórmula booleana, calcular la Transformada de Fourier Discreta $\rightarrow$ Número de objetos.
- Multiplicar enteros, verificar si un número es primo $\rightarrow$ Número de bits para representar un número en binario.
- Verificar si $G = (V, E)$ es conexa, plana, etc. $\rightarrow$ Más de un parámetro: $|V|$ y $|E|$.

Entonces, ¿Cómo queda eso del tiempo de ejecución?

Definición 3

El tiempo de ejecución de un algoritmo, esta dado por el número de pasos primitivos que ejecuta en una entrada particular.

Entonces, ¿Cómo queda eso del tiempo de ejecución?

Definición 3

El tiempo de ejecución de un algoritmo, esta dado por el número de pasos primitivos que ejecuta en una entrada particular.

Observaciones:

- Cada paso se ejecuta en una cantidad de tiempo constante.
- Cada linea de código toma un tiempo constante, pero no necesariamentel el mismo que otra linea.
- Tener cuidado con algunas lineas y enunciados del pseudocódigo.

Ejemplo

Primer análisis de tiempo de ejecución de Insertion-Sort

Ejemplo: Insertion-Sort

Realizar

Casos principales de análisis

Los tres casos principales para realizar un análisis de complejidad de tiempo de ejecución de un algoritmo son:

Casos principales de análisis

Los tres casos principales para realizar un análisis de complejidad de tiempo de ejecución de un algoritmo son:

Mejor caso: Se realiza el análisis bajo el supuesto de que se recibe una entrada que hara que el algoritmo ejecute el menor número de pasos (Ejemplos: elementos de un arreglo ordenados, elemento buscado es el primero del arreglo)

Casos principales de análisis

Los tres casos principales para realizar un análisis de complejidad de tiempo de ejecución de un algoritmo son:

Mejor caso: Se realiza el análisis bajo el supuesto de que se recibe una entrada que hara que el algoritmo ejecute el menor número de pasos (Ejemplos: elementos de un arreglo ordenados, elemento buscado es el primero del arreglo)

Caso promedio: Se realiza el análisis bajo la suposición de que la entrada representa un çaso típico"(¿Qué es eso?). Para realizarlo se utilizan análisis probabilístico, algoritmos aleatorios y se calculan tiempo esperado de ejecución.

Casos principales de análisis

Peor caso: Se realiza el análisis bajo el supuesto de que el algoritmo recibe una entrada que lo obligará a realizar el mayor número de pasos (Ejemplo: Elementos de un arreglo en orden inverso al deseado, busqueda de información inexistente). Este es el tipo de análisis que se aplica más comunmente.

A continuación

PROXIMAMENTE

A continuación

Ejercicios del capítulo

EJERCICIO 1

Encuentra un invariante del ciclo para el ciclo **while** *(lineas 5-8) del algoritmo Insertion-Sort, que sirva para escribir la prueba del Lema 2 de manera más formal. Prueba que este invariante se cumple en todas las iteraciones del ciclo.*

Ejercicios del capítulo

El siguiente es el pseudocódigo del método de ordenamiento de números BubbleSort:

```
function BubbleSort(A)
    for i = 0 downto A.length − 2 do
        for j = A.length - 1 downto i + 1 do
            if A[j] < A[j − 1] then
                swap(A[j], A[j − 1])
            end if
        end for
    end for
end function
```

Ejercicios del capítulo

EJERCICIO 2

Considera el peseudocódigo del programa BubbleSort:

1. *Implementa el método swap con una complejidad de tiempo de* $\Theta(1)$.
2. *Prueba la correctez del programa BubbleSort usando invariantes del ciclo.*

Ejercicios del capítulo I

El problema de Evaluar un polinomio $p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n$ esta dado por los siguientes datos:

Entrada: La secuencia de $n+1$ coeficientes $\langle a_0, a_1, \ldots, a_n \rangle$ de un polinomio $p(x)$ de grado n y un número z.

Salida: El resultado de evaluar $p(x)$ en el número z, es decir, la cantidad

$$\sum_{i=0}^{n} a_i z^i.$$

Ejercicios del capítulo II

EJERCICIO 3

Resuelve:

1. *Escribe pseudocódigo para la primera versión del método EVALUAR-INGENUO, el cual recibe como parámetros un arreglo A y un número z y devuelve como resultado la evaluación del polinomio*

$$A[0] + A[1]x + \cdots + A[n-1]x^{n-1} + A[n]x^n \qquad (2)$$

   en z.

2. *Prueba la correctez del método EVALUAR-INGENUO y da una cota asintótica para su tiempo de ejecución.*

Ejercicios del capítulo III

EJERCICIO 4

Continuamos con el problema de evaluar polinomios:

1. *Implementa el método EVALUAR-RH, el cual recibe como parámetros un arreglo A y un número z y devuelve como resultado la evaluación del polinomio (2) en z, pero utilizando la Regla de Horner.*
2. *Prueba la correctez del método EVALUAR-RH y da una cota asintótica para su tiempo de ejecución.*
3. *¿Cúal implementación es más eficiente para evaluar polinomios? ¿Porque? Justifica tus respuestas.*

Bibliografía I

Thomas H. Cormen y col. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.