Desarrollo de contenido curricular en el tema de Análisis y Diseño de Algoritmos Computacionales

Tesis para obtener el grado de Ingeniero en Computación

Presenta: José Julián Argil Torres

Director: M.I. Jorge Valeriano Assem

A mi madre por acompañarme y enseñarme desde siempre,

A mi hermano Mateo y mi hermana Sofía,

A Sol por apoyarme como nadie,

A Jorge por guiarme,

A mis compañeros del "Labo" por compartir tantas horas,

Y a muchos más que han estado y me han ayudado a llegar hasta este momento.

ÍNDICE

CAPÍTULO 1

Introducción

A lo largo de la preparación como Ingeniero se nos enseña que para llegar a una solución hay que estudiar el problema, quizá exista alguna herramienta que nos sirva para atacarlo, entonces deberemos buscarla, cuando la encontremos trataremos de aplicarla, generalmente estas herramientas serán matemáticas, esto suena como algo cotidiano cuando trabajamos con mecánica o quizá electrónica, ¿pero con la Computación? Generalmente llegamos a una solución, ¿pero es la mejor? ¿Podría haber sido mejor? ¿Cómo sabemos si es la mejor o por lo menos pretende serlo? Deberemos analizar nuestra solución.

En el capítulo 3 de esta tesis se explorarán algunas de las principales herramientas para el análisis de algoritmos, se verá probado que las ciencias básicas son tan básicas en la computación como en cualquiera otra de las ramas que se imparten en nuestra facultad.

Después del más reciente análisis que se ha hecho al plan de estudios, se ha decidido que el tema de los algoritmos es importante (aunque yo diría vital), para la formación de buenos Ingenieros en Computación, es por ello que nació la idea de escribir este trabajo, porque no existe tanto material del tema como nos gustaría, y el que hay en su mayoría se encuentra en otros idiomas, lo cual los hace inaccesible para algunos, además estos libros en su mayoría tratan el tema de los algoritmos desde un enfoque demasiado matemático/teórico que bajo ciertas circunstancias asustan al alumno Ingeniero que busca el conocimiento, de esta manera comencé a investigar, a aprender y desarrollar una idea y un enfoque, una forma de acercar un tema que considero tan importante hacia el resto de mis compañeros.

Este es el principal objetivo, escribir una teoría fundamentada en muchos y diversos libros de índoles más bien matemáticas y algunas veces computacionales, pero hacerlo con el enfoque que me da la formación de Ingeniero, redactarlo de tal manera que los demás como yo podrán entender de que les hablan cuando les dicen “obtén la big o de este algoritmo” y sin dudarlo podrán contestar “la obtengo y además sé qué significa el resultado”.

Pero todo este análisis solamente lo podemos aplicar si ya hemos planteado por lo menos algún bosquejo de solución, es por ello que decidí incluir también el capítulo 4 para las principales estrategias que podemos encontrar para solucionar un mundo de problemas, dándoles a todas las estrategias una descripción detallada, y en todos los casos una serie de ejemplos desarrollados de diferentes maneras y puestos a prueba bajo las herramientas matemáticas que para ese capitulo ya dominaremos.

El capítulo 5 trata sobre un Material para profesor y otro para alumno, quizá sea a través de este capítulos como se llegue a más gente, el Material para el profesor es una serie de presentaciones que un instructor podría utilizar para guiar un curso del tema en cuestión, y la parte para el alumno es una guía para el que esta escuchando, de tal manera que en todo momento pueda adentrarse más en el tema que su profesor le esté presentando. En otras palabras el capítulo 5 lleva toda la teoría desarrollada al aula de clase.

El material para ambos (profesor y alumno) se ha separado en módulos, que van desde el módulo 0 el cual va de la mano directamente con el capítulo 2 de esta tesis en el cual se introduce al lenguaje de programación C# (en ningún momento se pretende que este sea un curso del lenguaje sino una breve descripción de la sintaxis que pueda ser suficiente siendo que se conoce otro lenguaje similar como puede ser Java o incluso C++) luego el módulo 2

y 3 que van emparejados con el capítulo 3, finalmente los módulos 4, 5 y 6 que desarrollan a forma de curso el capítulo 4, al final de cada módulo se encuentra un laboratorio, que pondrá a prueba al estudiante con un ejercicio que deberá ser resuelto en un lenguaje de alto nivel, para este trabajo de tesis se ha escogido C# por las razones que en el capítulo 2 se detallan.

El capítulo 5 incluye además el desarrollo o guía de cómo se debe resolver cada uno de los laboratorios incluidos al final de cada módulo.

La solución para todos los laboratorios se ha incluido también.

Espero que al final de este trabajo de tesis, la importancia de analizar los algoritmos y de utilizar la estrategia adecuada para el problema que se desea solucionar quede clara.

CAPÍTULO 2

Microsoft Visual Studio 2005 y C#

2.1 Microsoft Visual Studio 2005 [14]

2.1.1 Introducción a Microsoft Visual Studio 2005

Para poder llevar a cabo los ejemplos y ejercicios propuestos, será necesario conocer ciertas características básicas del entorno de trabajo, en este caso de Microsoft Visual Studio 2005, el cual nos permite crear proyectos sin importar su complejidad o su tipo, ya que podemos crear tanto aplicaciones Web, como aplicaciones que correrán sobre una plataforma Windows.

Microsoft Visual Studio 2005 no es un lenguaje de programación, es un entorno integrado de desarrollo. Por lo tanto es una herramienta que nos facilita el llevar a cabo nuestra programación en el lenguaje de nuestra preferencia, siempre y cuando éste se encuentre soportado por el .NET Framework.

Visual Studio 2005 identificado en sus orígenes mediante el nombre código "Whidbey" (en referencia a la Isla Whidbey) fue oficialmente lanzado en línea en Octubre de 2005, solo unas semanas después llegó a las tiendas. Esta es la primera versión en la que Microsoft remueve la palabra ".NET" del título de su producto al igual que en todos sus productos lanzados en el mismo año, sin embargo el .NET Framework sigue siendo el principal objetivo, el cual fue actualizado a la versión 2.0. Visual Studio 2005 es la versión 8.0 en la serie de entornos de desarrollo producidos por Microsoft.

Entre varias mejoras, esta nueva versión agrega soporte a desarrollo de aplicaciones para 64 bits. Aún cuando el entorno esta disponible únicamente en versiones de 32 bits, permite compilaciones para x64 (AMD64 y EM64T) así como para IA-64 (Itanium).

Podemos encontrarlo en diferentes versiones, cada una incorpora más o menos cosas; la versión Professional nos sirve para desarrollar aplicaciones Web, aplicaciones para dispositivos móviles, para clientes inteligentes o aplicaciones Office, por otro lado podemos encontrar las versiones Standard y Express, indicadas para los que se inician en la programación, la segunda es además gratuita y se puede descargar desde el sitio oficial de Microsoft. [15]

Para el desarrollo de esta tesis se eligió la versión Professional, por lo que las imágenes presentadas pueden no ser las mismas que en las demás versiones.

2.1.2 Área de trabajo

Visual Studio 2005 es uno de los entornos de trabajo más intuitivos y robustos que podemos encontrar en el mercado; le agrega sencillez a todos los procesos complicados y repetitivos de la programación, haciendo que el desarrollador pueda preocuparse únicamente por escribir código necesario para llevar a cabo las tareas que requiere su aplicación.

En la figura 2.1 podemos observar la pantalla inicial del IDE, en la cual se nos presenta del lado derecho una serie de noticias extraídas de la MSDN (Microsoft Developers Network por sus siglas en inglés) y del lado izquierdo una lista con los proyectos

recientes, de tal manera que podemos abrirlos rápido, así como una serie de ayudas para comenzar a utilizar el entorno.

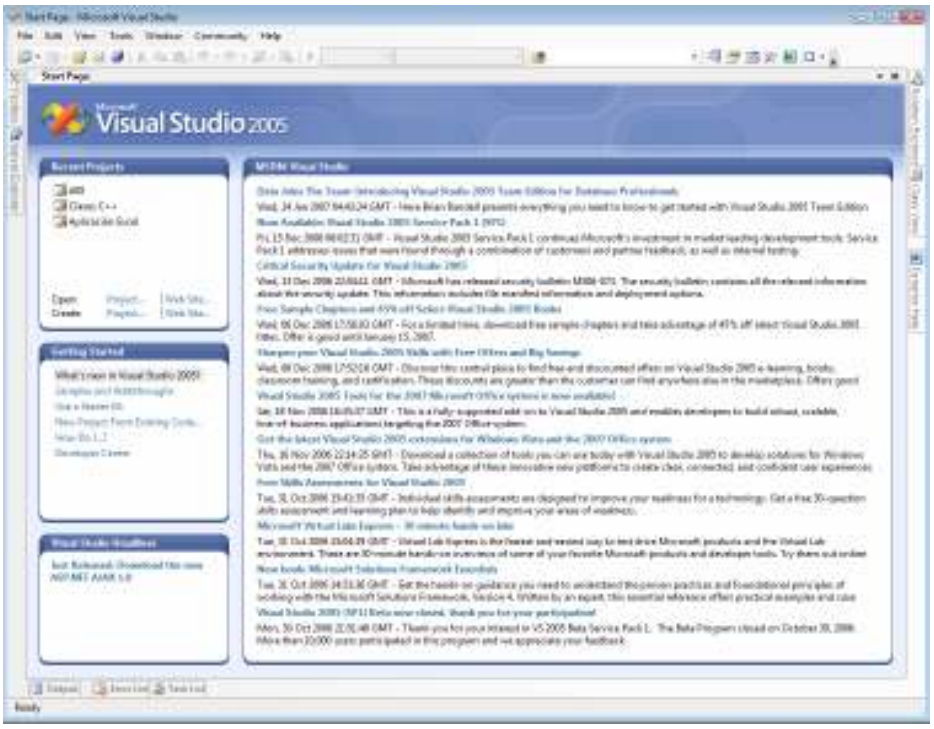


Figura 2.1 – Microsoft Visual Studio 2005 IDE

Cada uno de los controles presentados en la pantalla tienen asignado un nombre significativo y una posición en pantalla la cual podemos modificar para personalizarlo de acuerdo a nuestras preferencias. En la parte superior encontramos los menús, mediante los cuales podemos crear y modificar las propiedades de nuestros proyectos así como del área de trabajo.

2.1.3 Cómo crear un proyecto

Dar clic en el menú File, después en New y posteriormente en Project. Con lo cual aparecerá el dialog box de nuevo proyecto (Figura 2.2).

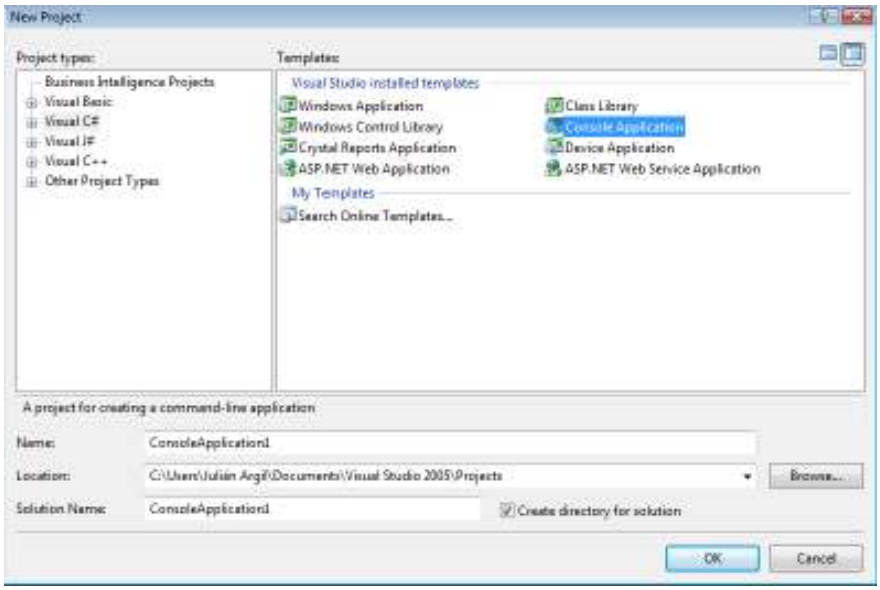


Figura 2.2 – Dialog box

Se puede observar que existe la posibilidad de crear proyectos para 4 lenguajes distintos, Visual Basic, Visual C#, Visual J# y Visual C++, todos ellos incorporados en la versión Professional de Visual Studio 2005.

Lo que sigue es seleccionar el lenguaje, para esta tesis será Visual C#, y seleccionar el tipo de proyecto, que en general se utilizará Windows Application o Console Application, pues nos sirven perfectamente para el desarrollo de algoritmos.

Finalmente debe asignarse un nombre a una primer aplicación, una ruta en la cual se guardará nuestro trabajo y un nombre a la solución, este último será el nombre del contenedor de aplicaciones, esto quiere decir que podemos tener una sola solución y trabajar con varias aplicaciones dentro de ella.

2.2 C# [12]

2.2.1 ¿Qué es C#?

C# es un lenguaje simple y moderno de programación orientada a objetos desarrollado por Microsoft como parte principal de su plataforma .NET Framework, posteriormente este lenguaje fue aprobado como un estándar por ISO y ECMA.

La sintaxis de C# se deriva del lenguaje C/C++ aunque contiene ciertas influencias de otros lenguajes como Java o Visual Basic.

2.2.2 ¿Por qué utilizar C#?

Al igual que en el mundo real, en el mundo de la programación existen diferentes herramientas, unas mejores que otras para realizar ciertas tareas. Existen lenguajes orientados a resolver ciertos problemas en particular; en el caso de los algoritmos, C# es una herramienta que nos provee de estructuras básicas, fáciles de programar y

comprender, así como un entorno de trabajo amigable que nos permite el preocuparnos únicamente por nuestro programa.

Es además un lenguaje muy intuitivo y similar a otros lenguajes muy populares como C, o Java lo que hace que sea un lenguaje ideal.

2.2.3 .NET Framework 2.0 [16]

El .NET Framework es un componente que podemos agregar a cualquier sistema operativo Windows. En la versión de Windows Vista se incorpora la versión 3.0 de forma predeterminada.

La versión 2.0 fue liberada a finales del año 2005. Esta mejora al .NET Framework se trajo junto con la actualización del IDE, Visual Studio 2005, incluye mejoras tales como la incorporación de Generics, que son plantillas como las utilizadas en C++ las cuales permiten tener un mejor control en los datos evitando así errores desde la compilación y no necesariamente hasta la ejecución de nuestras aplicaciones.

El .NET Framework mediante los namespaces incluidos en la librería de clases y en combinación con código producido por nosotros mismos, nos permite encontrar soluciones para diferentes problemas de la programación, como son soporte para aplicaciones Web, manejo de colecciones, trabajo con fuentes de datos, criptografía, comunicaciones mediante redes, expresiones regulares, etc.

Todos los programas escritos para el .NET Framework se ejecutan sobre un entorno de software, esto quiere decir que todos los requerimientos que llegue a necesitar la aplicación son administrados por otra aplicación de software llamada el CLR, de esta manera el desarrollador no tiene que preocuparse de las capacidades o características específicas del equipo sobre el cual se ejecutará la aplicación, ya que el CLR hace las veces de máquina virtual (Figura 2.3), proveyendo así de portabilidad a todos los programas escritos para .NET. Además de esto, el CLR también provee de mecanismos de seguridad, manejo de memoria, y una amplia capacidad para manejo de excepciones, evitando así los errores críticos de las aplicaciones.

Figura 2.3 – COM vs CLR

En si el .NET Framework es:

- Un entorno de ejecución
- Un conjunto de bibliotecas de funcionalidad (Class library)
- De libre distribución

.NET Framework se distribuye en 3 versiones:

- .NET Framework Redistributable Package
- .NET Framework SDK
- .NET Compact Framework

2.2.4 La evolución del .NET Framework [17]

La versión 1.0 del Framework fue liberada a principios del año 2002, e incluía también la versión 2002 de Visual Studio, así como varios lenguajes compatibles, como son Visual Basic y Visual C#.

La versión 1.1 fue liberada en el año 2003. Esta versión se introdujo junto con el Visual Studio 2003, junto con la primer versión del .NET Compact Framework así como junto con un nuevo lenguaje llamado Visual J#.NET.

La versión 2.0 del .NET Framework y del .NET Compact Framework fue liberada en el año 2005 junto con Visual Studio 2005.

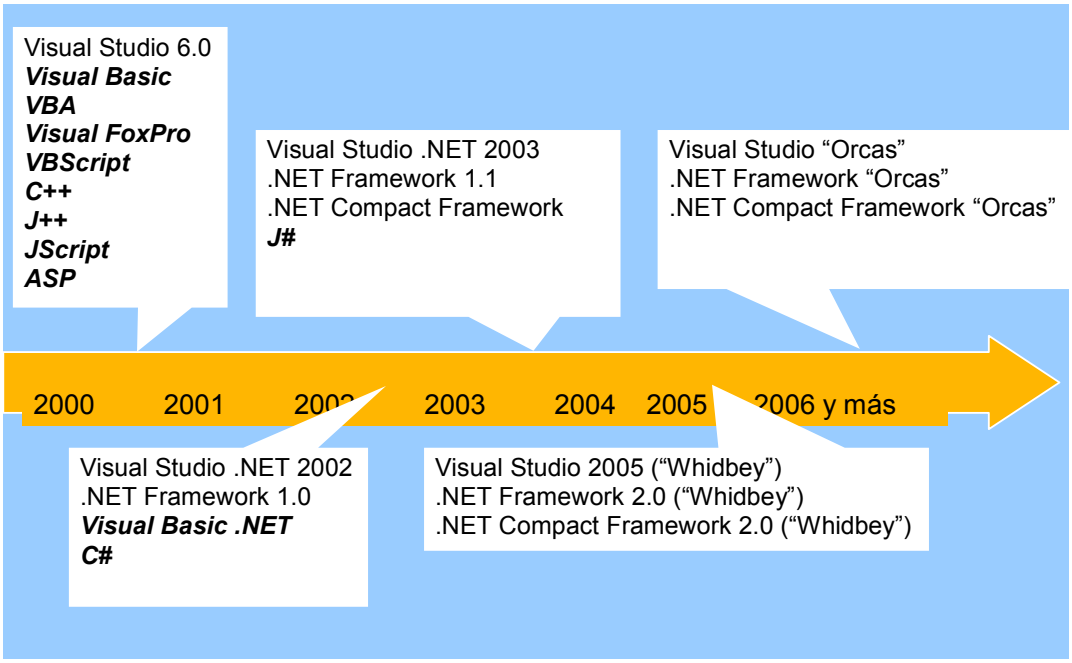


Figura 2.4 – Línea del tiempo

2.2.5 Arquitectura del .NET Framework

El .NET Framework esta compuesto de distintas capas (Figura 2.5).

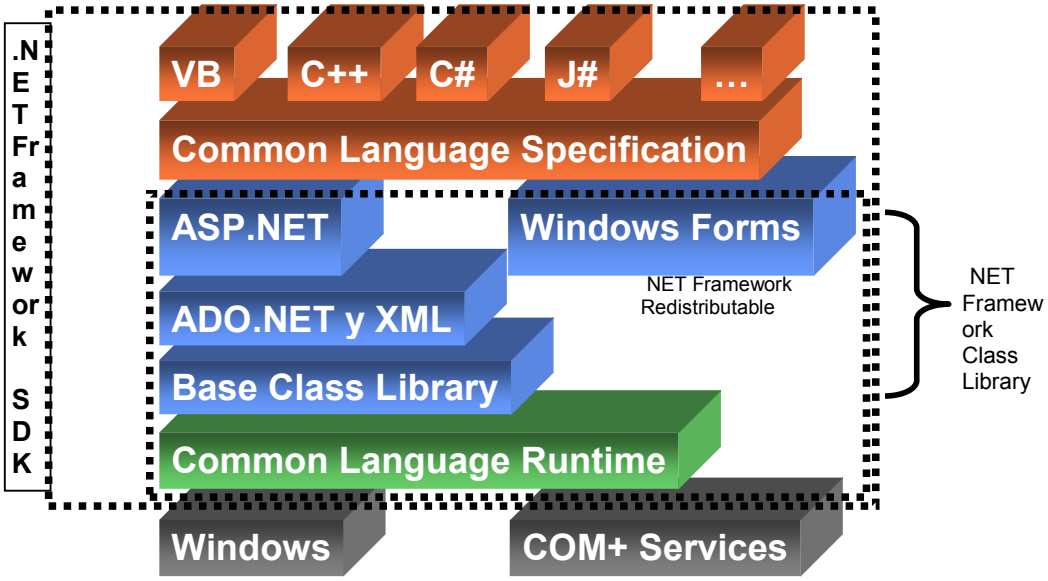


Figura 2.5 – Arquitectura

Esta arquitectura se monta sobre la familia de sistemas operativos Windows y sobre los servicios COM+.

En la parte más baja de la arquitetura se sitúa lo que se conoce como el Common Language Runtime o CLR por sus siglas en inglés, el cual se encarga como ya se mencionó antes de controlar la ejecución de las aplicaciones, por encima del CLR hablando de la arquitectura como una serie de capas, se sitúan un conjunto de librerías (.NET Framework Class Library), este conjunto de librerías se subdivide a su vez en 4 subcomponentes principales, el primero de ellos es la Base Class Library o BCL la cual contiene funcionalidades comúnmente utilizadas en las aplicaciones, como por ejemplo el trabajo con cadenas, programación multihilos, trabajo con matemáticas, manejo de colecciones como las pilas o las colas, etc. Después se encuentra ADO.NET el cual nos otorga una serie de clases para trabajar con archivos XML o realizar interacciones con manejadores de bases de datos relacionales como por ejemplo SQL Server 2005, dentro de este conjunto de librerías también encontramos ASP.NET y Windows Forms, el primero nos sirve para desarrollar aplicaciones Web y el segundo es una tecnología que nos permite desarrollar aplicaciones que se ejecutan directamente en el cliente y por lo tanto permiten un entorno mucho más rico y vistoso para el usuario. Finalmente y sobre todo esto se sitúan los compiladores y las herramientas de desarrollo para los lenguajes .NET. [20]

2.2.6 Introducción a la programación con C# [13]

Para poder comenzar, hay que conocer que tipos de datos podemos utilizar y que tipo de valores pueden almacenar, identificar cuales son los posibles nombres que podemos asignar a nuestras variables, aprender la sintaxis de las estructuras básicas del lenguaje, así como saber de qué partes esta compuesto un programa en C#.

2.2.6.1 Identificar las partes de un programa en C#

En la figura 2.6 se puede observar el código de un programa muy simple.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Capítulo_2_Introducción
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             Console.WriteLine("Introducción a la programación con C#");
12         }
13     }
14 }
```

Figura 2.6 – Ejemplo de programación con C#

De la línea 1 a las 3, se localizan las declaraciones de namespaces, hay que recordar que el .NET Framework nos da la facilidad de utilizar un conjunto de librerías de clases con soluciones a problemas comunes, la manera de acceder a ellas es mediante el uso de la palabra reservada *using* seguida del namespace que deseamos utilizar.

En la línea 5, se encuentra la declaración de un nuevo *namespace* seguido del nombre que deseamos asignarle (identificador), de esta manera nosotros podemos crear nuestros

propios namespaces e incluso exportarlos para compartirlos o utilizarlos en otros programas.

En la línea 6 hay una llave que abre, estas llaves en C# denotan el alcance de una cierta estructura, en este caso del namespace, van siempre acompañadas de una llave que cierra, en este caso se encuentra en la línea 14.

En la línea 7 se encuentra la declaración de una clase, C# por tratarse de un lenguaje orientado a objetos debe contener sus métodos y sus variables dentro de clases. En la línea 9 se puede ver un ejemplo de método, en este caso se trata del Main el cual representa el punto de entrada del programa, como puede verse esta contenido dentro de la clase "Program".

La línea 11 que se encuentra dentro del método Main, hace una llamada al método WriteLine que se encuentra dentro de la clase Console que se encuentra dentro del namespace System declarado en las primeras líneas. Este método sirve para arrojar una salida a la pantalla.

2.2.6.2 Palabras reservadas e identificadores

En C#, existen 77 palabras o identificadores reservados (Tabla 2.1), esto quiere decir que de todos los posibles nombres que podemos utilizar para nuestros propósitos no podemos elegir ninguno de estos. Estos identificadores son llamados palabras reservadas.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort
using	virtual	void	volatile
while			

Tabla 2.1 – Palabras reservadas

El significado y uso de algunas de estas palabras se verá en este capítulo.

2.2.6.3 Declaración y asignación de variables

Los identificadores que podemos asignar a nuestras variables no pueden coincidir con una palabra reservada; por otro lado, la sintaxis de C# no nos permite comenzar los identificadores con números o caracteres especiales, a excepción del guión bajo.

Las variables son espacios en los cuales almacenamos cosas de un cierto tipo, en C# existen diferentes tipos de cosas que podemos almacenar; cuando declaramos una variable debemos especificar un tipo de dato.

La sintaxis para declarar variables es:

tipo_de_dato identificador ;

En donde tipo_de_dato debe sustituirse con alguno de los soportados por el .NET Framework o por uno creado por nosotros mismos.

Una vez que hemos declarado una variable, podemos proceder a utilizarla, o mejor dicho asignarle algún valor para que esta lo almacene. Esto se hace mediante un operador, llamado el operador de asignación (Figura 2.7). En la primera línea puede verse la declaración de la variable y en la segunda la asignación del literal 5 a la variable. Todas las asignaciones en C# se hacen de derecha a izquierda.

```
1. int variable;
2. variable = 5;
```

Figura 2.7 – Declaración y asignación de una variable

C# contiene una serie de tipos de datos predefinidos llamados tipos de datos primitivos, la Tabla 2.2 lista lo más comunes así como los rangos de valores que podemos almacenar en ellos:

Tipo de dato	Descripción	Tamaño en bits	Rango*	Ejemplo de uso
int	Todo tipo de números enteros	32	<-> 231 hasta 231<->1	int variable; variable = 5;
long	Todo tipo de números enteros	64	<->263 hasta 263<->1	long variable; variable = 5L;
float	Números con punto flotante	32	$+-3.4x10^{38}$	float variable; variable = 0.5F;
double	Doble precisión con punto flotante	64	$+-1.7x10^{308}$	double variable; variable = 0.5;
decimal	Valores monetarios	128	28 figuras significativas	decimal variable; variable = 0.5M;
string	Secuencia de caracteres	16 bits por caracter	N/A	string variable; variable = “5”;
char	Un solo caracter	16	0 hasta 216<->1	char variable; variable = ‘5’;
bool	Booleanos	8	true o false	bool variable; variable = true;

Tabla 2.2 – Tipos de datos primitivos [14]
*** El valor 216 es 32,768; el valor 231 es 2,147,483,648 y el valor 263 es 9,223,372,036,854,775,808**

Una vez que hemos declarado una variable como cualquiera de los tipos de datos expuestos en la tabla anterior, antes de poder presentarlo en pantalla o de realizar operaciones aritméticas sobre ella, debemos inicializarla, en otras palabras asignarle un valor inicial, de lo contrario obtendremos un error en tiempo de compilación.

2.2.6.4 Operaciones aritméticas

En C# están soportadas todas las operaciones aritméticas comunes:

- + suma
- - resta
- * multiplicación
- / división
- % residuo

A estos símbolos mediante los cuales denotamos las operaciones, los llamamos operadores y a los valores que operan los llamamos operandos.

No todos los operadores se pueden utilizar en todos los tipos de datos. Por ejemplo, se pueden usar todos los operadores aritméticos en tipos de dato char, int, long, double o decimal, pero no todos en tipos string o bool.

```
int variable = 5;
variable = variable + 5;
```

Figura 2.8 – Ejemplo de suma

En este caso el operador suma modificaría el valor contenido dentro de variable, dejándolo como el resultado de sumar variable más 5, con lo que se obtendría 10.

La precedencia de los operadores en C# sigue las mismas reglas de la aritmética común, de tal manera que los operadores *, / y % tienen precedencia sobre + y -. Esto puede modificarse mediante el uso de paréntesis.

Existen además lo operadores de incremento y decremento los cuales nos servirán únicamente para modificar en 1 el valor de cualquier variable.

- variable++;
- ++variable;
- variable--;
- --variable;

Si el operador se encuentra antes o después indica si el incremento se hará antes o después de la lectura de la variable.

Existen además otros operadores llamados operadores compuestos, la siguiente tabla muestra cuales son y cómo funcionan a través de una analogía con los operadores aritméticos en su forma común.

Forma común	Operador compuesto
variable = variable + 5;	variable += 5;
variable = variable – 5;	variable -= 5;
variable = variable * 5;	variable *= 5;
variable = variable / 5;	variable /= 5;
variable = variable % 5;	variable %= 5;

2.2.6.5 Métodos

- Un método es una secuencia de sentencias/acciones.
- Cada método tiene un nombre y un cuerpo.
- El cuerpo del método contiene todas las sentencias que se ejecutan cuando el método es utilizado.
- Los métodos pueden recibir datos para procesar y regresar algún tipo de información.
- C# no soporta métodos globales por lo que todos deben ser escritos dentro de una clase.

```
tipoDeDato identificador (Lista de parámetros)
{
        cuerpo del método
}
```

Figura 2.9 – Sintaxis básica de un método

1. tipoDeDato debe ser el identificador de algún tipo de dato predefinido, especifica además que tipo de información será la que regrese el método. Puede ser cualquier tipo como int, string, o si el método no regresará información deberá especificarse void.
2. El identificador del método es mediante el cual lo podremos utilizar.
3. La lista de parámetros es opcional, y describe los tipos de datos y los valores que se le pasan al método.
4. El cuerpo del método incluye todas las líneas de código que se ejecutan cuando el método es llamado. Se delimita por las llaves que abren y cierran *{ ... }*.

La manera de regresar información desde un método es mediante la palabra reservada return seguida del valor que se desea regresar; el valor debe coincidir con el tipo de dato que se especifique en la declaración del método.

Para utilizar un método debemos llamarlo mediante su identificador, y debemos especificarle de ser necesario la lista de parámetros que requiera.

```
identificador ( Lista de parámetros ) ;
```

Figura 2.10 – Sintaxis para llamar a un método

2.2.6.6 Operadores relacionales y operadores condicionales lógicos

Sirven para saber si un valor es mayor, menor, igual o diferente que otro del mismo tipo mediante la formación de expresiones booleanas.

Operador	Significa	Ejemplo de expresión	Resultado si a=19
<	Menor que	a < 19 ;	false
<=	Menor o igual que	a <= 19;	true
>	Mayor que	a > 19;	false
>=	Mayor o igual que	a >= 19;	true
==	Igual que	a == 19;	true
!=	Diferente que	a != 19;	false

Tabla 2.3 – Operadores relacionales

Los operadores condicionales lógicos nos sirven para construir expresiones más complejas al permitirnos combinar operadores relacionales. Existen dos operadores condicionales, el primero de ellos es el operador OR denotado con || y el segundo es el operador AND denotado con &&.

2.2.6.7 Estructuras de decisión

a) IF

Se utilizan cuando queremos elegir entre ejecutar diferentes bloques de código dependiendo del resultado de una expresión booleana

```
1. ( a != 6)
2. ( a < 10 ) && (a > 0)
3. ((a >= b) || (b < 100)) && (a < 10)
```

Figura 2.11 – Ejemplo de expresiones booleanas

Sintaxis:

```
if ( Expresión booleana )
{
      Sentencias que se ejecutarán si la expresión es verdadera
}
else
{
      Sentencias que se ejecutarán si es falsa
}
```

Figura 2.12 - Sintaxis básica del IF

b) SWITCH

Se utiliza cuando podemos reducir la expresión booleana a una expresión común que nos permita decidir entre varios bloques variando únicamente el valor contra el cual se hace la comparación.

Sintaxis:

```
switch ( Expresión de control )
{
      case expresiónConstante:
          sentencias;
          break;
      case expresiónConstante:
          sentencias;
           break;
    …
    default:
           sentencias:
           break;
}
```

Figura 2.13 – Sintaxis básica del SWITCH

- La expresión de control es evaluada una sola vez.
- Las sentencias que se ejecutan son únicamente las que se encuentren en el case cuya "expresión Constante" coincide con el resultado arrojado por la expresión de control.
- Si ninguna de las expresiones constantes tiene un valor igual al de la expresión de control, se ejecutan las sentencias que contiene default.

2.2.6.8 Estructuras de iteración

a) WHILE

Esta estructura se puede utilizar para ejecutar una sentencia varias veces, en tanto una expresión booleana se cumpla.

Sintaxis:

```
while ( Expresión booleana )
{
        Sentencias
}
```

Figura 2.14 – Sintaxis básica del WHILE

La expresión booleana es evaluada y si es verdadera, las sentencias del cuerpo son ejecutadas, luego la expresión booleana es evaluada otra vez; si la expresión

sigue siendo verdadera, el cuerpo se ejecuta hasta que se vuelva a evaluar la expresión. Este proceso continúa hasta que la expresión booleana se evalúe falsa, entonces el while termina.

b) FOR

La sentencia for nos permite escribir una versión más formal de while al combinar la inicialización de una variable de control, la expresión booleana y la actualización de la variable de control en un mismo lugar.

Sintaxis:

```
for ( inicialización; expresión Booleana; actualización)
{
        Sentencias
}
```

Figura 2.15 – Sintaxis básica del FOR

La inicialización de la variable o variables de control ocurre una sola vez al inicio del ciclo for. Si la expresión booleana es evaluada verdadera, las sentencias se ejecutan. La variable de control es actualizada antes de que la expresión booleana sea reevaluada. Mientras la expresión booleana se mantenga verdadera se seguirán ejecutando las sentencias del cuerpo.

c) DO – WHILE

Las estructuras WHILE y FOR evalúan su expresión booleana al inicio del ciclo. Esto significa que si la expresión booleana se evalúa falsa desde el principio, el cuerpo del ciclo no se ejecuta ni una vez.

La sentencia DO – WHILE es diferente, su expresión booleana es evaluada después de la primer iteración, por lo que el cuerpo es ejecutado por lo menos una vez.

Sintaxis:

```
do
{
        Sentencias;
} while ( Expresión booleana );
```

Figura 2.16 – Sintaxis básica del DO – WHILE

d) FOREACH

La estructura FOREACH declara una variable de iteración que automáticamente toma el valor de cada elemento de un cierto arreglo o contenedor. La sintaxis del FOREACH es mucho más declarativa; es decir expresa mucho mejor la intención del código a diferencia de un simple FOR.

La estructura FOREACH es la mejor para iterar a través de arreglos y de colecciones.

Sin embargo en algunos casos hay que utilizar un FOR:

- Una estructura FOREACH siempre itera a través de todo el arreglo o colección. Si se quiere iterar sobre una porción o pasar sobre ciertos elementos, es más fácil utilizar una estructura FOR.
- Una estructura FOREACH siempre itera desde el índice 0 al índice final. Si se quiere iterar de atrás hacia adelante, se necesita un FOR.
- Si el cuerpo del ciclo necesita conocer el índice del elemento en lugar de solo su contenido, se necesita usar un FOR.
- Si se necesita modificar los elementos del arreglo, se necesita usar un FOR. Esto es porque la variable de iteración es solamente una copia del valor contenido en el arreglo.

Sintaxis:

```
foreach ( tipo identificador in contenedor )
{
        Sentencias
}
```

Figura 2.17 – Sintaxis básica FOREACH

El tipo de identificador debe ser del mismo que los elementos almacenados en el contenedor. Contenedor puede ser por ejemplo un arreglo o una colección.

2.2.6.9 Manejo de errores y excepciones

Es un hecho que las cosas malas suceden. Los errores pueden ocurrir en cualquier momento de la ejecución de nuestros programas. Para ello C# incorpora una estructura que nos permite probar ciertas líneas de código y si se detecta un error permitirnos corregirlo y evitar que se convierta en algo crítico.

Para poder hacer esto, tenemos que hacer básicamente 2 cosas:

- Escribir el código dentro de un bloque *try*. De esta manera cada línea de código dentro del bloque se tratará de ejecutar.
- Escribir tantos bloques *catch* como sean necesarios inmediatamente después del bloque *try* para poder interceptar cualquier posible condición de error.

Sintaxis:

```
try
{
        Sentencias a probar/ejecutar
}
catch ( Excepción a interceptar )
{
        Corrección del posible error
}
catch ( Segunda posible excepción )
{
        Corrección del posible error
}
…
finally
{
        Sentencias que siempre se deberán
        ejecutar
}
```

Figura 2.18 – Sintaxis básica try – catch

Si se detecta una condición de error en el bloque try, la ejecución del programa salta inmediatamente al primer bloque catch cuya excepción coincida con dicha condición.

Cuando el flujo del programa se ve modificado, es obvio que algunas líneas de código pueden llegar a no ser ejecutadas, en algunas ocasiones ciertas sentencias son vitales y no podemos permitir que no se lleven a cabo, con este propósito se utiliza el bloque *finally*.

2.2.6.10 Clases y objetos

Las clases se declaran utilizando la palabra reservada *class*.

```
class Clase
{
        Métodos, propiedades, campos, eventos, delegados, y clases anidadas se
        escriben aquí
}
```

Figura 2.19 – Clases

Una vez que declaramos una clase podemos utilizarla como utilizaríamos cualquier otro tipo de dato con la diferencia de que las clases necesitan ser "construidas" mediante el uso de la palabra reservada *new* y la utilización de un constructor.

Dentro de las clases escribiremos todo el código de nuestros programas.

Los métodos, propiedades, campos y demás contenido de las clases puede estar afectado por modificadores de acceso. Estos modificadores permiten o no el acceso a dichos elementos desde clases exteriores. Los modificadores de acceso básicos son:

- Public
- Protected
- Private

Public provee de la máxima libertad de acceso, mientras que private provee de la máxima seguridad. En cuanto a protected se comportará de diferente manera dependiendo de si se trata de una clase derivada o no desde la cual se trata de tener acceso. Si no se especifica un modificador de acceso, por omisión el compilador tomará private.

2.2.6.11 Enumeraciones

Las enumeraciones son un set de constantes en forma de lista. Cada enumeración tiene un tipo de dato asociado el cual puede ser cualquier tipo de entero, este sirve como índice para acceder a cualquier elemento dentro de la lista. Por omisión el índice se almacena en un int y el primer elemento de la lista tiene asociado el índice 0, para cada elemento sucesivo el índice se incrementa en 1.

Sintaxis:

```
enum identificador
{
        Lista de elementos
}
```

Figura 2.20 – Sintaxis básica de enum

La lista de elementos son una serie de literales separados entre si por comas.

Una vez que hemos declarado una enumeración, podemos utilizarla de la misma manera que utilizaríamos cualquier otro tipo de dato, por ejemplo un string. Esto quiere decir que podemos crear variables del tipo de nuestra enumeración.

Ejemplo:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Capítulo_2_Introducción
{
    class Program
    {
        enum Ejemplo
        {
            PrimerElemento,
            SegundoElemento,
            TercerElemento
        }
        static void Main(string[] args)
        {
            Ejemplo variable; //creación de la variable
            variable = Ejemplo.SegundoElemento; //asignación
            Console.WriteLine(variable);
        }
    }
}
```

Figura 2.21 – Declaración y uso de enumeraciones

Este pedazo de código imprimiría en consola: “SegundoElemento”.

2.2.6.12 Estructuras

Las estructuras tienen una sintaxis muy similar al de las clases.

```
struct identificador
{
        Métodos, propiedades, etc.
}
```

Figura 2.22 – Estructuras

Las diferencias básicas entre una clase y una estructura son:

- Las estructuras no pueden tener constructores default.
- En las estructuras si se hace un constructor alternativo, se deben inicializar todos los campos.
- En el caso de las estructuras los campos no pueden ser inicializados desde que son declarados.

Para poder utilizar las estructuras, al igual que con las clases se debe crear una variable a partir de la estructura y se debe utilizar la palabra reservada new para poder llamar al constructor y así inicializar la variable de tipo estructura.

Ejemplo:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Capítulo_2_Introducción
{
    class Program
    {
        struct Estructura
        {
            public Estructura(int valor) //Constructor
            {
                campo = valor;
            }
            public int Metodo() //Método
            {
                return campo;
            }
            int campo; //Campo
        }
        static void Main(string[] args)
        {
            Estructura variable = new Estructura(5);
            Console.WriteLine(variable.Metodo());
        }
    }
}
```

Figura 2.23 – Declaración y uso de estructuras

2.2.6.13 Arreglos

Un arreglo es una secuencia no ordenada de elementos. Todos los elementos en el arreglo tienen que ser del mismo tipo. Los elementos de un arreglo se ubican en bloques contiguos de memoria y son accedidos utilizando un índice de tipo entero.

Para declarar un arreglo se necesita especificar el tipo de dato, seguido de un par de paréntesis cuadrados, seguido de un identificador (Figura 2.24).

```
tipoDeDato[ ] identificador ;
```

Figura 2.24 – Sintaxis básica de un arreglo

Se pueden crear arreglos de estructuras, enumeraciones y clases, no únicamente de tipos primitivos.

Para poder utilizar los arreglos, hay que crear después de declarar el identificador una instancia mediante el uso de la palabra reservada *new*.

```
1.tipo[ ] identificador = new tipo[ tamaño del arreglo ]; //unidimensional
2.tipo[ , ] identificador = new tipo[ tamaño , tamaño ]; //bidimensional
```

Figura 2.25 – Declaración de arreglos multidimensionales

Los arreglos pueden inicializarse también mediante el uso de enumeraciones sin tener que especificar el tamaño:

```
tipo[ ] identificador = {1,2,3,4};
```

Figura 2.26 – Definición del arreglo mediante una enumeración

Para acceder un elemento individual de un arreglo, se debe especificar el índice del elemento que se requiere entre paréntesis cuadrados. El índice es un valor entero.

- Lectura de un índice del arreglo:

```
int variable;
variable = arreglo[ índice que se desea leer ] ;
```

Figura 2.27 – Leyendo del arreglo

- Escritura en un índice del arreglo:

```
int variable = 23;
arreglo [ índice en el que se desea escribir ] = variable;
```

Figura 2.28 – Escribiendo en el arreglo

2.2.6.14 Colecciones

Los arreglos son muy útiles, pero tienen limitantes. De cualquier manera, los arreglos son solo una manera para almacenar elementos de un mismo tipo. El .NET Framework provee de muchas clases que sirven para coleccionar elementos de otras maneras más especializadas. Estas clases son llamadas Colecciones, y las podemos encontrar en el namespace System.Collections.

Arreglos contra colecciones:

- Un arreglo declara el tipo de los elementos que almacena, una colección no. Esto es porque las colecciones almacenan sus elementos como objetos.
- Una instancia de arreglo tiene un tamaño determinado el cual no puede modificarse. Una colección tiene un tamaño dinámico.
- Un arreglo es una estructura de datos de lectura/escritura (no hay manera de crear un arreglo de solo lectura), sin embargo es posible utilizar colecciones en un modo de solo lectura utilizando el método ReadOnly.

Existen 5 clases de colecciones muy comunes:

a) ArrayList

Es una clase muy útil para ordenar elementos en forma de arreglo. Sobre todo en ocasiones en que un arreglo convencional es muy restrictivo:

- Si queremos modificar el tamaño del arreglo, tenemos que crear una copia.
- Si queremos remover elementos del arreglo, tenemos que hacer una copia de los elementos y luego moverlos todos 1 posición.
- Si queremos insertar un elemento en el arreglo, debemos mover los elementos 1 posición para dejar un espacio libre.
- Podemos remover elementos de un ArrayList simplemente utilizando el método Remove. El ArrayList esta implementado para que automáticamente se reordenen sus elementos.
- Podemos agregar elementos al final del ArrayList, utilizando el método Add. El ArrayList modifica automáticamente su tamaño.
- Podemos insertar elementos a la mitad del arreglo utilizando el método Insert. ArrayList modifica su tamaño si es necesario.

Ejemplo:

```
using System;
using System.Collections; //necesario para uso de colecciones
…
ArrayList numeros = new ArrayList(); //creación del ArrayList
…
//llenar el ArrayList
foreach(int numero in new int[12]{1,2,3,4,5,6,7,8,9,10,11,12})
        numeros.Add(numero);
…
//remover el primer elemento que su valor sea 7
numeros.Remove(7); //indice 6
//remover el elemento en el índice 6
numeros.RemoveAt(6);
…
//iterar en los 10 elementos restantes
for(int i = 0; i != numeros.Count ; i++)
{
        int numero = (int)numeros[i]; //cast de object a int
        Console.WriteLine(numero);
}
…
//iterar utilizando un foreach
foreach (int numero in numeros) //no hace falta un cast
        Console.WriteLine(numero);
```

Figura 2.29 - ArrayList

b) Queue

La clase Queue implementa un mecanismo FIFO (first-in first-out).

Un elemento es insertado en la cola en la parte trasera y es removido de la parte delantera.

Ejemplo:

```
using System;
 using System.Collections;
 …
 Queue numeros = new Queue();
 …
 //llenando
 foreach (int numero in new int[4]{1,2,3,4})
 {
        numeros.Enqueue(numero);
        Console.WriteLine(numero + "ha entrado a la cola");
 }
 …
 //iterando para obtener los elementos de la cola
 foreach(int numero in numeros)
 {
        Console.WriteLine(numero);
 }
 ...
 //vaciando la cola
 while(numeros.Count != 0)
 {
        int numero = (int)numeros.Dequeue();
        Console.WriteLine(numero + "ha salido de la cola");
 }
```

Figura 2.30 - Queue

c) Stack

Implementa un mecanismo LIFO (last-in first-out). Un elemento entra al stack en la cima (operación push) y la abandona desde la cima (operación pop).

Ejemplo:

```
using System;
using System.Collections;
…
Stack numeros = new Stack();
…
//llenando la pila
foreach(int numero in new int[4]{1,2,3,4})
{
        numeros.Push(numero);
        Console.WriteLine(numero + "ha entrado a la pila");
}
…
//iteranco
foreach(int numero in numeros)
        Console.WriteLine(numero);
…
//vaciando la pila
while(numeros.Count != 0)
{
        int numero = (int)numeros.Pop();
        Console.WriteLine(numero + "ha salido de la pila");
}
```

Figura 2.31 – Stack

d) Hashtable

Los arreglos y los ArrayList nos proveen una manera de utilizar índices para acceder a un elemento. Hay ocasiones en que podemos querer acceder a la información de los elementos mediante índices que no sean de tipo int. En otras palabras utilizar arreglos asociativos. La clase Hashtable provee de esta funcionalidad al mantener internamente dos arreglos de objetos, uno para las llaves que se están utilizando como índices y otro para los valores que utilizamos para crear otros tipos de índices.

Cuando insertamos un par de llaves en una Hashtable, automáticamente detecta que llave pertenece a cada valor, y nos permite recuperar el valor asociado a cada llave. Hay algunas consecuencias al utilizar la clase Hashtable:

- No pueden haber llaves duplicadas.

- Cuando utilizamos un foreach para iterar en un Hashtable, obtenemos un DictionaryEntry. La clase DictionaryEntry provee de acceso a la llave y a los valores de los elementos en ambos arreglos, a través de la propiedad Key y Value.

Ejemplo:

```
using System;
using System.Collections;
…
Hashtable edades = new Hashtable();
…
//llenando
edades[“Julian”] = 24;
edades[“Sol”] = 20;
edades[“Mateo”] = 12;
…
//iterando utilizando un foreach
foreach (DictionaryEntry elemento in edades)
{
        string nombre = (string)elemento.Key;
        int edad = (int)elemento.Value;
        Console.WriteLine(“Nombre: {0} Edad: {1}”), nombre, edad;
}
```

Figura 2.32 – Hashtable

e) SortedList

La clase SortedList es muy similar a la clase Hashtable, ya que permite asociar llaves con valores. La principal diferencia es que las llaves del arreglo siempre son ordenadas.

Cuando insertamos una llave/valor en una SortedList, la llave es insertada dentro del arreglo de llaves en el lugar correcto para mantener así las llaves ordenadas. El valor es insertado en el arreglo de valores en el mismo índice. La clase SortedList automáticamente asegura que las llaves y los valores estén alineados, aún cuando se agregue o remueva elementos. Esto significa que podemos insertar llaves/valores en una SortedList en cualquier orden que queramos; y ellas siempre serán ordenadas de acuerdo al valor de su llave.

Como la clase Hashtable, una SortedList no puede contener llaves duplicadas. Cuando utilizamos un foreach para iterar, recibimos un DictionaryEntry, los objetos vendrán ordenados de acuerdo a la propiedad Key

Ejemplo:

```
using System;
using System.Collections;
…
SortedList edades = new SortedList();
…
//llenando
 edades[“Julian”] = 23;
edades[“Sol”] = 19;
edades[“Mateo”] = 11;
…
//iterando
foreach(DictionaryEntry elemento in edades)
{
        string nombre = (string)elemento.Key;
        int edad = (int)elemento.Value;
        Console.WriteLine(“Nombre: {0} Edad: {1}”, nombre, edad);
}
```

Figura 2.33 - SortedList

CAPÍTULO 3

Fundamentos de análisis y diseño de algoritmos

3.1 Algoritmos

3.1.1 ¿Qué son?

Un algoritmo se puede ver como cualquier procedimiento computacional bien definido el cual toma uno o varios valores de entrada y produce uno o varios valores de salida. Por lo tanto un algoritmo es esa secuencia de pasos computacionales que se tienen que llevar a cabo para resolver un problema específico obteniendo una salida a partir de cierta entrada.

De esto deriva, que si un problema puede ser resuelto algorítmicamente, significa que podemos escribir un programa de computadora que lo resuelva.

3.1.2 Historia [3]

Los algoritmos se ocupan de cierta manera desde hace más años de los que nos imaginamos. Un ejemplo de esto es la evaluación de potencias de un número desconocido X, para lo cual hace más de dos mil años se desarrolló un algoritmo que reducía el número de cálculos necesarios de forma considerable. Como este ejemplo podemos encontrar muchos otros. Sin embargo, hasta antes del siglo XX, el estudio de los algoritmos se había hecho de forma muy dispersa, es decir, había mucha gente trabajando pero no se estaba desarrollando una teoría general.

En 1834 Charles Babbage ya había concebido pero de forma mecánica el concepto de un algoritmo mediante su "Máquina Analítica". A diferencia de la "Máquina Diferencial" creada años antes, su Máquina Analítica pretendía resolver problemas de índoles que no fueran solamente cálculos de tablas logarítmicas, es decir, sería una máquina que pudiera realizar cualquier tipo de cálculo, convirtiéndose en una máquina de propósito general. Esta máquina sería programable mediante tarjetas perforadas, por lo tanto siguiendo la idea de un algoritmo, cada tarjeta representaría un paso en la solución de un problema, de tal manera que el conjunto de tarjetas para resolver dicho problema representaría un algoritmo computacional.

El concepto matemático formal de algoritmo, fue formulado hasta la década de 1930, incluso antes de la llegada de las computadoras. Durante este proceso de formalización, se trabajó mucho con la definición de que cosas podrían ser resueltas algorítmicamente y que cosas no serían posibles. En este proceso, la mayor influencia sobre el desarrollo de las teorías algorítmicas vino del matemático inglés Alan Turing, quien describió un modelo llamado la Máquina de Turing, en el cual estableció el mecanismo mediante el cual las computadoras podrían resolver problemas. Además de esto, Alan Turing también describió de forma teórica el paradigma de los programas almacenados, el cual representa la base para las computadoras de propósito general de hoy en día. Sin embargo, John von Neumann fue quien demostró formalmente la practicidad de introducir las instrucciones y los datos que estas procesan en la memoria de una computadora.

En la actualidad, si se usan los algoritmos apropiados, pueden resolverse problemas que alguna vez fueron imposibles. Ejemplos de esto son la robótica y la inteligencia artificial que implica, el cómputo gráfico, la solución a grandes problemas mediante sistemas computacionales, incluso la comunicación mediante redes implica el uso de los algoritmos correctos.

3.2 Complejidad

3.2.1 ¿Para qué y por qué?

Para resolver un problema en particular mediante un algoritmo, la mayoría de las veces, sino es que todas, tendremos más de una opción, es decir podremos resolver el problema de más de una forma; si podemos analizar estos algoritmos, podremos mejorarlos y en el mejor de los casos elegir el que resuelva nuestro problema.

Para analizar los algoritmos, podremos tomar como criterios a medir diferentes cosas, entre las cuales vamos a encontrar la complejidad. La complejidad no es más que la medición de la cantidad de trabajo que realiza un algoritmo. Existen otros posibles factores que podemos medir como por ejemplo el espacio utilizado o la claridad del código. [4]

La complejidad o cantidad de trabajo realizado por un algoritmo, no depende del equipo sobre el cual se esté ejecutando, ni del lenguaje sobre el cual se esté programando, por lo que para este trabajo, la complejidad es el factor a medir para el análisis de algoritmos.

3.2.2 ¿Qué se debe medir?

Medir la cantidad de trabajo de un algoritmo podría llevar a la idea de que se medirán tiempos de ejecución, sin embargo esto no es lo mejor, pues los tiempos de ejecución de un cierto programa dependerán en gran medida de la computadora sobre la que estemos trabajando además del lenguaje sobre el cual hayamos programado.

Lo que buscamos medir como cantidad de trabajo es aquello que nos sirve como punto de comparación entre dos o más algoritmos que realizan la misma tarea, y además que no sea dependiente de la computadora o del lenguaje empleado. [4]

Elegiremos como características a medir las operaciones que realiza un cierto algoritmo elegido. De todas las operaciones que lleva a cabo, deberemos escoger solamente algunas que nos darán pauta para poder llevar a cabo las comparaciones. Por ejemplo, en el caso de un algoritmo de búsqueda podríamos elegir como operación la comparación que debe realizar dicho algoritmo entre el valor que buscamos y cada posición de la colección de elementos. [4]

Sin embargo, la cantidad de trabajo que realiza un algoritmo variará en gran medida de acuerdo al tamaño de las entradas que le proporcionemos, por ello es que debemos elegir bien las operaciones que utilizaremos para analizar la complejidad.

3.2.3 Análisis de complejidad de peor caso

Como ya se mencionó antes, la cantidad de trabajo o complejidad del algoritmo, dependerá en gran medida del tamaño de las entradas que le proporcionemos. Es por ello que además de las operaciones, debemos saber elegir una medida para el tamaño de las entradas. De aquí podemos poner como ejemplo el mismo que antes, si necesitamos realizar una búsqueda dentro de una colección, la medida del tamaño debería ser el número de elementos de dicha colección.

Teniendo esta información, podremos realizar un primer análisis de algoritmos, el más utilizado para esto, es el llamado análisis de complejidad de peor caso porque nos proporciona una cota superior del trabajo que realiza el algoritmo.

La definición que utilizaremos para la complejidad de peor caso es el siguiente:

Sea D_n el conjunto de entradas de tamaño n para el problema en consideración, y sea I un elemento de D_n. Sea t(I) el número de operaciones seleccionadas que el algoritmo ejecuta con la entrada I. Definimos la función W de la siguiente manera:

$$W(n) = máx\{t(I) \mid I \in D_n\}$$

Figura 3.1 – Peor caso [3]

Esta función representará el número máximo de operaciones que realiza un algoritmo seleccionado para cualquier entrada de tamaño n.

3.2.4 Análisis de complejidad promedio

En algunas ocasiones una mejor medida de complejidad podría ser el trabajo promedio que realiza un algoritmo en lugar del peor caso. Esto se hace mediante la siguiente definición:

Sea P(I) la probabilidad de que se presente la entrada I. Entonces, el comportamiento promedio del algoritmo será:

$$A(n) = \sum_{I \in D_n} P(I)t(I)$$

Figura 3.2 – Complejidad promedio [3]

La dificultad del análisis de complejidad promedio esta en el hecho de que se necesita utilizar un valor de probabilidad el cual no puede ser calculado de forma analítica como sería por ejemplo el valor de t(I). Por lo tanto, el valor que se le asigne a P(I) deberá ser supuesto, por ejemplo que todas las entradas tienen la misma probabilidad de aparecer.

3.2.5 Ejemplo de análisis de complejidad de peor caso y complejidad promedio [5]

Tomaremos como ejemplo el de la búsqueda de un cierto elemento dentro de una colección que no se encuentra ordenada:

Algoritmo: Búsqueda de forma secuencial

Entrada: arreglo, elementoBuscado

Salida: Verdadero si se encuentra el elemento buscado, Falso si no se encuentra.

```
n ← Número de elementos que contiene el arreglo
for i←0 to n-1 do
        if elementoBuscado == arreglo en su posición i
                return Verdadero termina el algorítmo
        endif
endfor
return Falso termina el algoritmo
```

1. Análisis de complejidad de peor caso:

   Para saber cual será el resultado de este análisis elegimos como operación la comparación que se realiza dentro del ciclo entre lo que buscamos y cada posición dentro del arreglo y como medida de entrada el número de elementos del arreglo. Teniendo esto en cuenta, es fácil saber que en el peor de los casos el elemento buscado se encontrará al final de dicho arreglo, por lo tanto el máximo número de comparaciones que se realizarán será n, es decir el número total de elementos. Por lo tanto W(n) = n.

2. Análisis de complejidad promedio

   Para llevar a cabo este análisis deberemos tomar en consideración que el elemento buscado si se encuentra dentro del arreglo no ordenado.

   Recordando la ecuación mencionada para este análisis, deberemos nombrar una variable X, la cual nos permitirá realizar la sumatoria desde que X es 0 hasta n-1 (el tamaño del arreglo menos uno), por lo tanto 0 <= X < n. Para la ecuación debemos elegir un valor de probabilidad de éxito, es decir cuál es la posibilidad de que se presente el valor buscado, como sabemos que nuestra entrada será de tamaño n y asumimos que los valores no se repiten dentro del arreglo, la probabilidad será $P(I)=\frac{1}{n}$. Finalmente el número de

operaciones que se realizarán, será t(I) = i+1, puesto que el máximo valor de i será n-1, sin embargo el número de operaciones será n = i + 1.

De esto deriva que:

$$A(n)=\sum_{i=0}^{n-1} P(I)t(I)$$

Sustituyendo P(I) y t(I) tenemos que:

$$A(n) =\sum_{i=0}^{n-1} \frac{1}{n}(i+1)$$

De esto, sabemos que $\frac{1}{n}$ es una constante, por lo que podemos sacarlo de la sumatoria, y además sabemos que:

$$\sum_{i=1}^{n} i =\frac{n(n+1)}{2}$$

Por lo que podemos aplicarlo para:

$$\sum_{i=0}^{n-1} i+1$$

De tal manera que obtenemos:

$$A(n) =\frac{1}{n}\left(\frac{n(n+1)}{2}\right)$$

Finalmente:

$$A(n) =\frac{n+1}{2}$$

3.3 Análisis asintótico

3.3.1 Necesidad

Medir la cantidad de operaciones realizadas nos permite distinguir entre dos algoritmos siempre y cuando las cantidades sean drásticamente diferentes. Sin embargo, si dos algoritmos parecen realizar el mismo número de operaciones, será necesario hacer un

análisis un poco más completo, es decir tendremos que compararlos de tal manera que solo tomemos en cuenta las entradas grandes. Al establecer esto, y siguiendo con la notación antes utilizada, querrá decir que lo que nos importará será n, es decir nos fijaremos en el orden de las funciones con las cuales estemos trabajando y dejaremos de lado cualquier otro término que no cambie de forma sustancial la magnitud de dicha función. [2]

De esta manera podremos tener como resultado otra función, la cual nos dará de forma aproximada la eficiencia de la función original, sin embargo para nosotros este aproximado será suficiente cuando estemos trabajando con entradas grandes, como se estableció anteriormente. A esta medida de eficiencia bajo estas condiciones, se le llama Análisis Asintótico de Complejidad.

Ejemplo:

$$\text{Sea } f(n)=n^3+10{,}000n+1{,}000{,}000$$

Para n = 10, evaluando

$$f(10)=10^3+10{,}000(10)+1{,}000{,}000=1{,}000+100{,}000+1{,}000{,}000$$

Se puede observar que para valores pequeños de n, la constante 1,000,000 es el valor más grande de la función.

Para n = 100, evaluando

$$f(100)=100^3+10{,}000(100)+1{,}000{,}000=1{,}000{,}000+1{,}000{,}000+1{,}000{,}000$$

Se puede observar que en algún momento los 3 términos tiene el mismo valor y por lo tanto contribuyen de igual manera a la función.

Para n = 1000, evaluado

$$f(1{,}000)=1{,}000^3+10{,}000(1{,}000)+1{,}000{,}000=1{,}000{,}000{,}000+10{,}000{,}000+1{,}000{,}000$$

Cuando el valor de n comienza a ser grande, la contribución que realizan el segundo y tercer término a la función comienza a ser mucho menor que el que realiza el primer término, esto es debido al crecimiento cúbico que tiene n^3. De tal manera que el único término que nos interesaría para entradas grandes sería el primero.

3.3.2 Notaciones Big O, Big Ω y Big Θ [5]

Cuando de hacer análisis asintótico se trata, se utilizan básicamente tres notaciones:

- Big O
  - O(g) son todas aquellas funciones que no crecen más rápidamente que g

- Big Ω
  - Ω(g) son todas aquellas funciones que crecen por lo menos tan rápidamente como g
- Big Θ
  - Θ(g) son todas aquellas funciones que crecen tan rápidamente como g, es decir la intersección entre Big O y Big Ω

Para definir cada una de estas notaciones, tendremos que establecer que el orden de las funciones esta restringido únicamente a valores reales $f(n): N \to R^*$ en donde N es el conjunto de los naturales con $N=\{0,1,2,...\}$ y R* es el conjunto de los números reales no negativos, de tal manera que existe un entero n_0 (que depende de f) tal que $f(n) > 0$ para todos los $n > n_0$. Denotaremos como F a todo el conjunto de funciones que cumplan con lo anterior.

- Definición de O(g):

  Sea una función $g(n) \in F$, definimos $O(g(n))$ como el conjunto de todas las funciones $f(n) \in F$ que tienen la propiedad de que para una constante real c y una constante entera no negativa n_0, y para todo $n \geq n_0$,

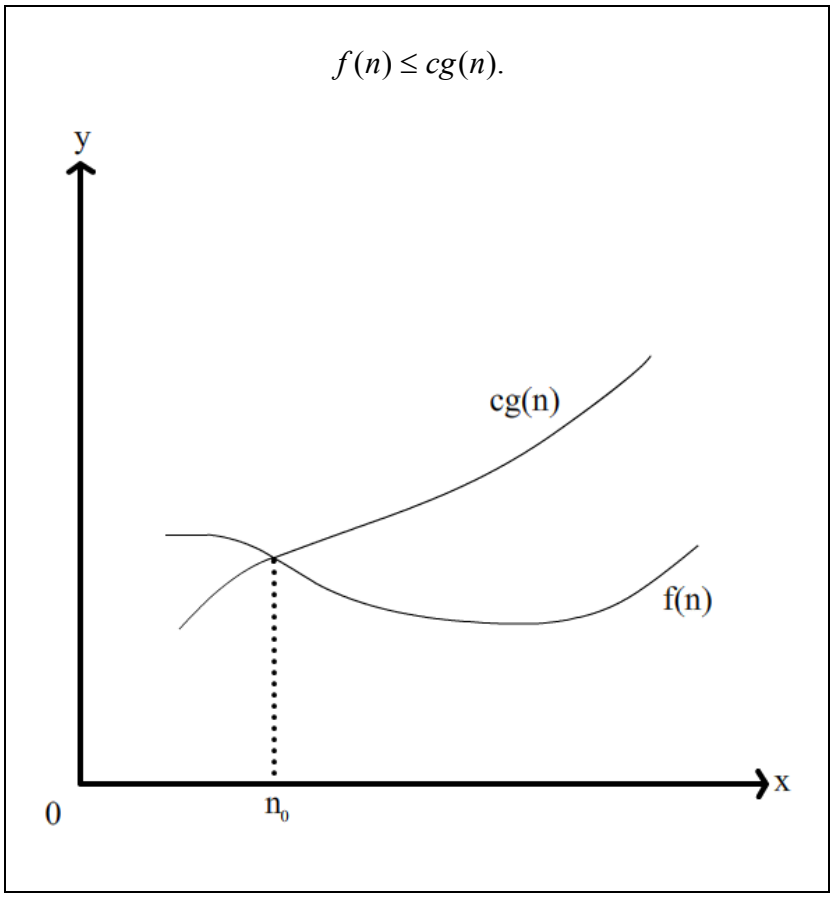


Figura 3.3 – O(g) [3]

Si $f(n) \in O(g(n))$ entonces $f(n)$ es Big O de $g(n)$. [5]

- Definición de Ω(g):

  Sea una función $g(n) \in F$, definimos $\Omega(g(n))$ como el conjunto de todas las funciones $f(n) \in F$ que tienen la propiedad de que para una constante real c y una constante entera no negativa n_0, y para todo $n \geq n_0$,

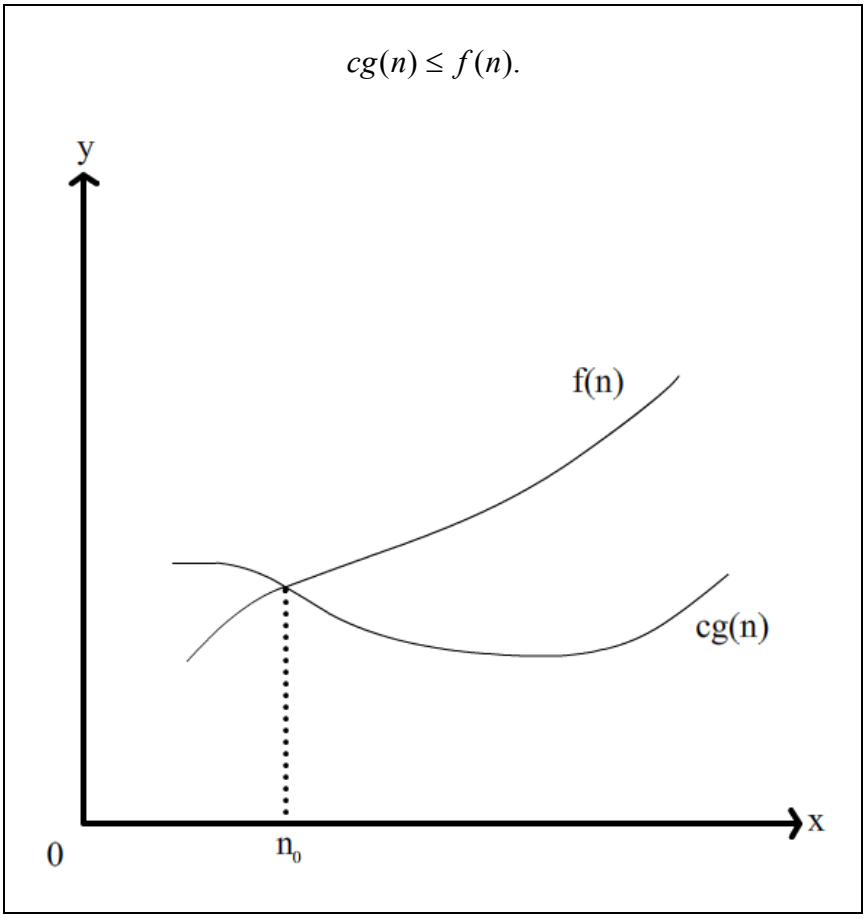


Figura 3.4 - Ω(g) [3]

Si $f(n) \in \Omega(g(n))$ entonces $f(n)$ es Big Ω de $g(n)$. [5]

- Definición de Θ(g):

  Sea una función $g(n) \in F$, definimos $\Theta(g(n))$ como el conjunto de todas las funciones $f(n) \in F$ que tienen la propiedad de que para dos constantes reales c_1 y c_2 y una constante entera no negativa n_0, y para todo $n \geq n_0$,

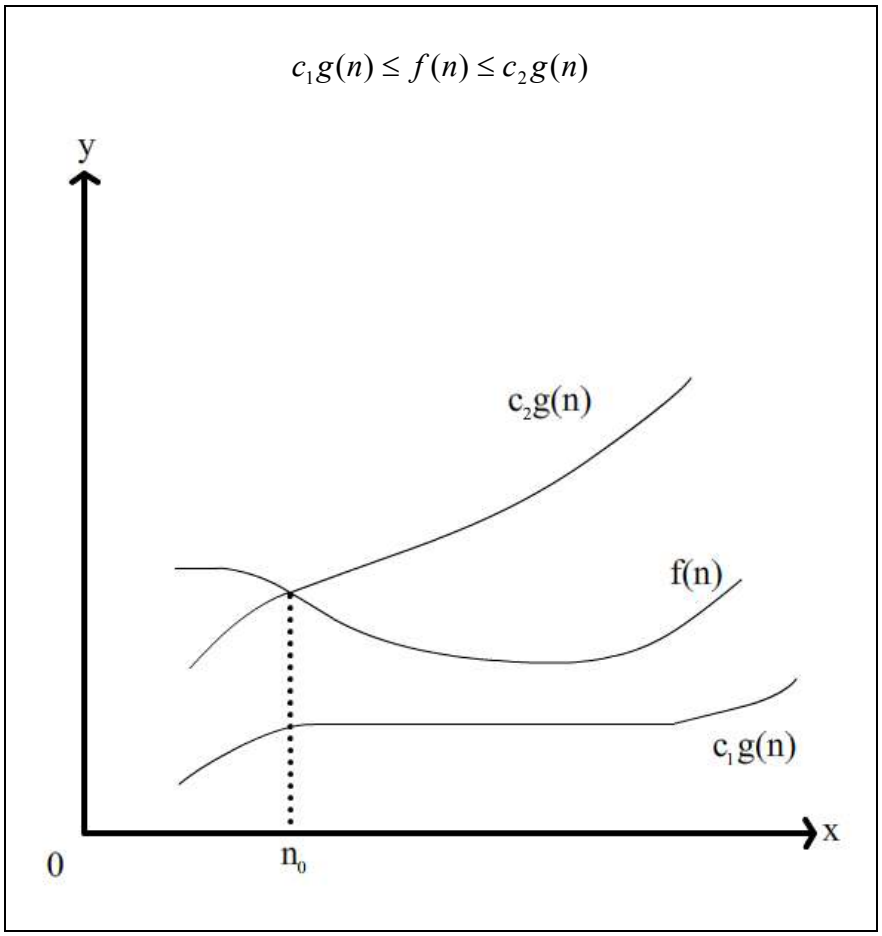


Figura 3.5 - Θ (g) [3]

.

Es decir, $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Si $f(n) \in \Theta(g(n))$ entonces $f(n)$ es Big Θ de $g(n)$. [5]

3.3.3 Big O

Recordando la definición de Big O:

> Sea una función $g(n) \in F$, definimos $O(g(n))$ como el conjunto de todas las funciones $f(n) \in F$ que tienen la propiedad de que para una constante real c y una constante entera no negativa n_0, y para todo $n \geq n_0$, si $f(n) \in O(g(n))$ entonces $f(n)$ es Big O de $g(n)$.

Por lo tanto sabemos que existen una serie de constantes (c, n y n_0), pero en ningún momento se ha dicho como calcular estos valores. Como se verá a continuación, pueden

haber muchísimos pares de c y n, dada una función f(n), una manera de obtenerlas es utilizando la función que estamos analizando y la definición de Big O.

Del ejemplo anterior, tenemos:

Sea $f(n)=n^3+10{,}000n+1{,}000{,}000 \in O(n^3)$ como se mostró al final de la sección 3.3.1

Sabemos por la definición que:

$$n^3+10{,}000n+1{,}000{,}000 \leq cn^3$$

Por lo tanto despejando la inecuación obtenemos:

$$1+\frac{10{,}000}{n^2}+\frac{1{,}000{,}000}{n^3} \leq c$$

De aquí podemos asignar valores, de tal manera que formemos los siguientes pares:

n_0	$n_0=1$	2	3	4	100	101	200	1000	tiende a ∞
c	$c \geq 1010001$	127501	38149	16251	3	2.95	1.37	1.0110	tiende a 1

Para elegir el mejor par de c y n_0, lo que tenemos que observar es en que punto uno de los factores de nuestra ecuación se convierte en el mayor y se mantiene siendo el mayor aún si seguimos aumentando n_0. Los únicos términos en nuestro ejemplo que nos sirven para esto son n^3 y $10{,}000n$, en base a lo que vimos en secciones anteriores, queremos hallar los valores de c y n, para los cuales se cumple que:

$$n^3 > 10{,}000n$$

Esto se cumple a partir de que $n > 100$, por lo tanto obtenemos que el mejor par en nuestra tabla es:

c = 2.95 y n = 101

Es decir, si $cg(n)$ es mayor o igual que $f(n)$ desde n=101, entonces c tiene que ser al menos 2.95.

Recordando la definición que se dio para O(g), podemos ver que la utilidad de hallar los posibles pares de c y n es mantener siempre la condición de Big O, la cual nos dice que g(n) es casi siempre más grande o igual que f(n) mientras g(n) sea multiplicada por una constante c correcta y n sea mayor que una constante n_0. Esto queda más claro con la Figura 3.6.

En la siguiente gráfica se comparan dos valores de $cg(n)$ contra la función $f(n)$ que es la cual estamos analizando en búsqueda de su Big O.

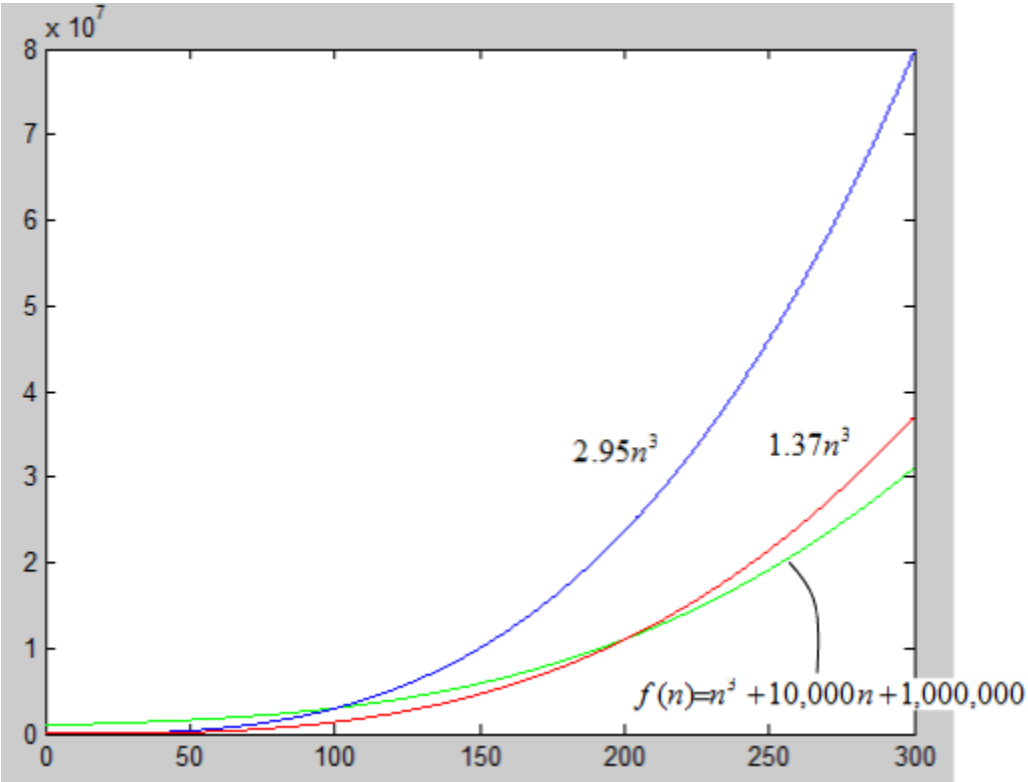


Figura 3.6 – Funciones g(n) – f(n)

Se puede observar en la gráfica que el punto en el cual se cruzan las funciones $g(n)$ con $f(n)$ coincide con el valor de n para la constante c elegida.

De la figura 3.6 podemos concluir que para una función dada podemos encontrar múltiples funciones que cumplan con la definición de Big O, sin embargo para eliminar todo ese conjunto de posibilidades siempre nos interesará la menor, en este caso $O(n^3)$.

Sabiendo esto, ¿cómo podemos identificar si una función es Big O/ Ω/ θ de otra función? Podemos emplear algunas herramientas.

Para Big O, podemos utilizar el siguiente Lema:

Una función $f(n) \in O(g(n))$ si el $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, incluido el caso cuando el límite es 0.

Figura 3.7 – Big O [5]

Si el límite del cociente de f(n) entre g(n) sí existe y no es infinito, podemos decir que g(n) crecerá más rápidamente que f(n). Pero si el límite es ∞ entonces f(n) sí crece más rápidamente que g(n).

Ejemplo:

Sean $f(n) = 100n + 1000$ y $g(n) = n^2 + 10$. Demostrar que $f(n) \in O(g(n))$.

Para $n \geq 110$, sabemos que se cumple que $f(n) \leq cg(n)$ en donde $c = 1$, por lo tanto $f(n) \in O(g(n))$.

Resolviéndolo mediante el lema especificado en la Figura 3.7, podemos ver que:

$$\lim_{n \to \infty} \frac{100n + 1000}{n^2 + 10} = 0$$

Por lo tanto $f(n) \in O(g(n))$.

Para el análisis asintótico O(g(n)) podemos utilizar también otro teorema que aplica a límites:

Sea $f(n)$ y $g(n)$ funciones diferenciables, con derivadas $f'(n)$ y $g'(n)$ respectivamente, tales que

$$\lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty$$

Entonces

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$$

Figura 3.8 – Regla de L'Hôpital [5]

Ejemplo:

Sean $f(n) = \lg_2 n + 1000$ y $g(n) = n^2 + 10$. Demostrar que $g(n) \notin O(f(n))$.

Para lo cual tenemos que:

$$\lim_{n\to\infty} \frac{n^2 + 10}{\lg_2 n + 1{,}000}$$

Sabemos que para convertir de una base logarítmica a otra, podemos utilizar el siguiente lema:

$$\log_c x = \frac{\log_b x}{\log_b c}$$

Con lo que tenemos:

$$\lim_{n\to\infty} \frac{n^2 + 10}{\frac{\ln(n)}{\ln(2)} + 1{,}000}$$

Si multiplicamos todo por ln(2):

$$\lim_{n\to\infty} \frac{(n^2 + 10)\ln(2)}{\ln(n) + 1{,}000\ln(2)}$$

Aplicando regla de L'Hôpital:

$$\lim_{n\to\infty} \frac{2n\ln(2)}{\frac{1}{n}}$$

Finalmente:

$$\lim_{n\to\infty} 2n^2 \ln(2) = \infty$$

3.3.4 Propiedades de Big O, Big Ω y Big Θ [4]

Estas notaciones tienen una serie de propiedades que nos pueden servir para facilitarnos el análisis de algoritmos. En todas las propiedades, suponemos que las funciones $f(n), g(n), h(n) : N \to R*$.

1. (Transitividad) Si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, entonces $f(n) \in O(h(n))$. Lo mismo aplica para Ω y Θ.
2. $f(n) \in O(g(n))$ si y sólo si $g(n) \in \Omega(f(n))$.
3. Si $f(n) \in \Theta(g(n))$, entonces $g(n) \in \Theta(f(n))$.
4. Si $f(n)$ es $O(h(n))$ y $g(n)$ es $O(h(n))$, entonces $f(n) + g(n)$ es $O(h(n))$.
5. Si $f(n) = cg(n)$, entonces $f(n)$ es $O(g(n))$.

3.3.5 Jerarquía [3]

Algunas complejidades se pueden identificar de cierta forma como menores que otras, tal es el caso de una complejidad de tipo linear O(n) y una complejidad cúbica $O(n^3)$. Sabemos que O(n) es de menor complejidad que $O(n^3)$.

Siguiendo la definición que se encuentra en la figura 3.9, podemos construir una serie de conjuntos y construir un diagrama que nos permita identificar la jerarquía de diferentes complejidades.

Dadas las funciones $f(n)$ y $g(n)$, podemos decir que $f(n)$ es de orden menor que $g(n)$ si $O(f(n))$ esta contenido en $O(g(n))$, es decir, $O(f(n)) \subset O(g(n))$.

Figura 3.9 - Definición

Al realizar los análisis Big O, Ω y Θ nos damos cuenta de que ciertas funciones se presentan con mucha frecuencia como resultado:

- 1
- logn
- $\sqrt{n}$
- n
- nlogn
- n^2
- n^3
- 2^n
- $n2^n$

- n!

A partir de ellas podemos generar un diagrama de Venn (figura 3.9) que ilustre la jerarquía mediante la identificación de ciertas funciones como subconjunto de otras, es decir:

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n2^n) \subset O(n!)$$

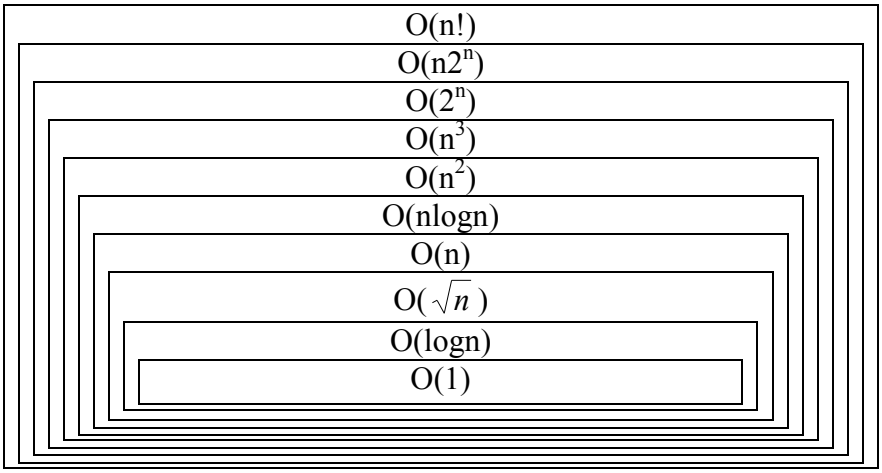


Figura 3.10 – Diagrama de Venn

Sabiendo esto, podríamos decir que si una función es O(n) por lo tanto también será $O(n^2)$ debido a que si $f(n) \in O(g(n)) \Leftrightarrow O(f(n)) \subseteq O(g(n))$. Sin embargo en el análisis de algoritmos lo que nos interesa es identificar la complejidad de menor orden para cada algoritmo.

A continuación se presenta una tabla que refleja el tiempo que se tardaría en ejecutar cierta cantidad de instrucciones en una computadora que pueda ejecutar 1 instrucción por microsegundo. Se pretende que con esta tabla quede claro por qué algunas complejidades son menores que otras:

N	**10 instrucciones**	**100 instrucciones**	**1,000 instrucciones**	**10,000 instrucciones**	**100,000 instrucciones**
$O(1)$	1 μs	1 μs	1 μs	1 μs	1 μs
$O(\log n)$	3 μs	7 μs	10 μs	13 μs	17 μs
$\sqrt{n}$	3 μs	10 μs	31 μs	100 μs	316 μs
n	10 μs	100 μs	1,000 μs	10,000 μs	100,000 μs
$n \log n$	33 μs	664 μs	10,000 μs	133,000 μs	1.6 *seg*
n^2	100 μs	10,000 μs	1 *seg*	1.7 min	16.7 min
n^3	1 *ms*	1 *seg*	16.7 min	11.6 *día*	31.7 *año*

2^n	$1.024\,ms$	$4*10^{16}\,año$	$3.39*10^{287}\,año$	…	…
$n2^n$	$10.24\,ms$	$4*10^{18}\,año$	…	…	…
$n!$	$4\,seg$	$2.95*10^{144}\,año$	…	…	…

Queda claro que complejidades mayores requieren mucho más tiempo para realizar el mismo número de instrucciones. En algunos casos el tiempo es tan grande que deja de ser un algoritmo que pueda utilizarse. [4]

3.3.6 Problemas NP-Completos [11]

Existen dos clases de problemas que nos conviene definir antes de poder decir lo que son los problemas NP-Completos: La clase P y la clase NP.

La clase P solo se puede definir para problemas de decisión, es decir problemas en los cuales solo existen dos posibles respuestas: sí o no. Además P es la clase de problemas que se encuentran polinómicamente acotados: Problemas en los cuales su complejidad en el peor caso se puede definir mediante un polinomio del tamaño de las entradas. Los problemas que se encuentren dentro de esta clase, serán en muchas ocasiones eficientes, y podrán ser resueltos en un tiempo razonable, los problemas que no pertenezcan a esta clase por el contrario en su mayoría serán problemas que no se puedan resolver. Una propiedad de los problemas de la clase P, es que podemos obtener la solución a un problema a partir de la combinación de algoritmos para problemas más fáciles siempre y cuando los algoritmos para estos problemas también se encuentren polinómicamente acotados. Además de todo esto, los problemas P son resueltos por algoritmos determinísticos, es decir el conjunto de algoritmos que dada una entrada solo existe un posible camino para determinar el siguiente paso que deberá dar el algoritmo.

La clase NP también se define para los problemas de decisión, que deben poder ser polinómicamente acotados y a diferencia de la clase P, los algoritmos NP son no-determinísticos, es decir son algoritmos que utilizan algún tipo de operación que cuando se presenta una decisión hacen una conjetura para saber cuál es el siguiente paso que se debe tomar. La clase P es un subconjunto de NP, pues los algoritmos determinísticos son aquellos algoritmos no determinísticos que no utilizan decisiones no determinísticas.

Un problema es NP-Completo si se puede resolver mediante un algoritmo polinómico no determinístico, y además se puede reducir cualquier problema NP a este problema NP-Completo:

Un problema se puede reducir a otro problema, si existe alguna función que al ser procesada con instancias del primer problema, devuelva instancias del segundo problema, es decir, para cada instancia del primer problema habrá una contraparte del segundo. La reducción no es simétrica, es decir un problema A puede ser reducido a un problema B, pero lo contrario no es necesariamente verdadero. Reducir un problema nos sirve pues si podemos encontrar una instancia del problema B para cualquier instancia del problema A,

entonces una solución eficiente del problema B puede ser eficientemente transformada en una solución al problema A.

Un problema A en NP es NP-Completo si todos los problemas en NP son reducibles a A. Por lo tanto todos los problemas NP-Completo son computacionalmente equivalentes.

Los problemas de NP-Completo son los problemas más difíciles de NP. En la actualidad cualquier algoritmo que forme parte del conjunto NP-Completo, empleará tiempo exponencial para hallar una solución (ver tabla en la sección 3.3.5), por lo que generalmente para resolver este tipo de problemas se emplean soluciones probabilísticas o simplemente se hacen aproximaciones.

3.4 Ejemplos

3.4.1 Ciclo

Tomamos como primer ejemplo el siguiente ciclo:

```
for i←0 hasta n-1 hacer
        j ← arreglo[i] "arreglo en su posición i"
endfor
```

La operación que utilizaremos para hacer el análisis asintótico y obtener Big O, serán las asignaciones que se realizan en la inicialización y durante el ciclo.

Sabemos que i se inicializa en 0 y se incrementa de 1 en 1 hasta n-1.

Primero se inicializa la variable i (i←0), después el ciclo itera n veces (de 0 a n-1), y durante cada iteración se asigna un nuevo valor a i y a j. De tal manera que al final del ciclo for se deberán haber hecho 1 + 2n asignaciones.

Si $f(n) = 1 + 2n$ y mediante el uso de las definiciones, sabemos que el único factor que a entradas grandes contribuirá al valor de la función es n, de tal manera que $f(n) \in O(g(n))$ en donde $g(n) = n$. Se puede escribir como que su complejidad asintótica es $O(n)$.

3.4.2 Ciclo anidado

Para este ejemplo lo que analizaremos es un ciclo dentro de otro ciclo, es decir un ciclo anidado:

```
for i←0 hasta n-1 do
        for j← hasta i do
                x ← arreglo[j] "arreglo en su posición j"
        endfor
endfor
```

Una vez más utilizaremos las asignaciones como nuestra operación básica.

Al igual que en el ejemplo anterior el for exterior se ejecuta *n* veces e inicializa para ello a i, solo que en esta ocasión en cada iteración del ciclo exterior se ejecuta un ciclo interior y se hacen asignaciones a *i*, *j* y *x* por lo tanto habrán 1+3n asignaciones. El ciclo interior se ejecuta *i* veces por cada *i* que en cada iteración exterior puede tomar valores entre 1 y n-1, además en cada iteración del ciclo interior se hacen asignaciones a *x* y *j*, en esta ocasión no podemos considerar la asignación inicial de *j*, pues esa ya la estamos considerando como parte del ciclo exterior, por lo tanto para el ciclo interior tenemos $\sum_{i=1}^{n-1} 2i$, ¿por qué una sumatoria? A diferencia del ejemplo anterior, en esta ocasión el ciclo se ejecutará tantas veces como iteraciones tenga el ciclo exterior.

Finalmente tenemos que habrá $1+3n+\sum_{i=1}^{n-1} 2i$ asignaciones, esto se puede ver también como:

$$1+3n+\sum_{i=1}^{n-1} 2i =$$

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

Por lo tanto la complejidad asintótica de este ejemplo es $O(n^2)$.

3.4.3 Ordenamiento por inserción

El algoritmo de este ejemplo es el que sigue:

```
for j←1 to n do
        auxiliar ← arreglo[j] "arreglo en su posición j"
        i ← j – 1
        while (i >= 0) and (arreglo[i] > auxiliar)
                arreglo[i + 1] ← arreglo[i]
                i ← i – 1
        endwhile
        arreglo[i + 1] ← auxiliar
endfor
```

El ciclo exterior se ejecuta n-1 veces. Las asignaciones a *auxiliar*, *i* y *arreglo* (dentro del for y afuera del while) se llevan a cabo exactamente n – 1 veces. Las asignaciones dentro del ciclo while son las que buscan la posición en la cual debe colocarse el elemento que se esta ordenando, el número de asignaciones a *arreglo* e *i*, va a depender de cuantas comparaciones deba realizar para encontrar dicha posición, para nuestro análisis nos

conviene pensar en el peor de los casos que se presentaría si el arreglo que tratamos de ordenar esta invertido a como lo queremos, es decir si buscamos ordenarlo de forma ascendente se encuentra en forma descendente o viceversa, en este caso cada asignación dentro del while se va a llevar a cabo exactamente *j-1* (observar que la variable con la que se compara en el while es i que es igual a *j – 1*) veces para cada elemento a ordenar. Por lo tanto tenemos que habrá $1+3(n-1)+\sum_{j=1}^{n} j-1$, si se observa la sumatoria siempre va a empezar en 1 y a terminar en n porque se debe realizar para todo el ciclo for exterior. Por lo tanto obtenemos que:

$$1+3(n-1)+\sum_{j=1}^{n} j-1 = 1+3n-3+\frac{(n)(n-1)}{2} = -2+3n+\frac{1}{2}(n^2-n) \in O(n^2)$$

CAPÍTULO 4

Principales estrategias de diseño de algoritmos

4.1 "Divide y vencerás"

4.1.1 Introducción

Divide y vencerás es una estrategia de diseño de algoritmos muy poderosa. Se basa en tomar un problema grande y por lo tanto difícil de resolver como entidad y dividirlo en pedazos del mismo tipo del problema de acuerdo a un cierto criterio. Una vez que se ha hecho esta división, se procede a resolver cada uno de los pedazos por separado mediante métodos recursivos o métodos directos, en algunas ocasiones la división proporciona pedazos pequeños de los cuales ya se conoce su respuesta. Cuando se tiene la solución de cada pedazo lo único que resta es juntarlos de alguna manera para así obtener la respuesta al problema completo. [1]

Como se dijo, la estrategia de Divide y vencerás significa seguir una serie de pasos básicos, por ello es que si queremos resolver un problema específico de esta manera, nuestro algoritmo deberá contener ciertos métodos casi de forma obligatoria.

Estos métodos harán lo siguiente:

- Resolver el problema de una forma directa si es que la entrada que se recibió es lo suficientemente pequeña como para no requerir una división.

- Llevar a cabo la división en caso de que la entrada que se recibió sea lo suficientemente grande.

- La combinación de los resultados de los diferentes pedazos que hayan surgido de la división.

De tal manera que nuestro algoritmo en una forma muy generalizada se verá así: [5]

Algoritmo: Divide y vencerás

Entrada: E serán los datos de entrada, n será el tamaño de E

Salida: Solución al problema específico

if n <= un tamaño predefinido como suficientemente pequeño
    return Solución directa del problema a partir de E
else
    Dividir el problema en pedazos más pequeños E_1 hasta E_i a partir de E
    Soluciones parciales = Solución a cada pedazo desde E_1 hasta E_i
    Solución al problema = Combinación de las soluciones parciales
    return Solución al problema

Algunos de los algoritmos que resuelven típicamente sus problemas mediante la estrategia de divide y vencerás son los algoritmos de ordenamiento; por ejemplo Quicksort y Mergesort.

4.1.2 Quicksort [5]

Fue inventado por C.A.R. Hoare en 1962, es uno de los primeros en implementar la estrategia "divide y vencerás", y se puede considerar como uno de los algoritmos más rápidos para realizar la tarea de ordenamiento.

Este algoritmo basa su funcionamiento en la división de una colección de datos en pequeñas colecciones creadas a partir de la entrada original y posteriormente resuelve cada uno de estos pedazos recursivamente.

Previamente se vio que un algoritmo del tipo divide y vencerás debe tener ciertas partes casi de forma obligatoria, además cabe recalcar que el trabajo comúnmente se centra en dividir o combinar, y entre estos el trabajo no se divide de forma equitativa, alguno de los dos hará la mayor parte del trabajo mientras que el otro tendrá un trabajo muy sencillo.

En el caso del algoritmo de ordenamiento Quicksort la mayor parte de su trabajo se basa en la división mientras que la combinación de los resultados de los pedazos divididos es relativamente simple.

Quicksort es un algoritmo que funciona en su lugar, es decir no necesita de memoria adicional para realizar el ordenamiento.

La estrategia de este algoritmo se basa en dividir una lista grande, o un arreglo grande A[primero, último] en base a un pivote cualquiera x. A[primero, último] se ordena de forma tal que antes del pivote x solo quedan elementos que cumplen con ser menores que x, y después del pivote todos los elementos son mayores que x.

Un ejemplo para el arreglo de números enteros sería [8,6,3,4,22,9,1,11], si se toma como pivote al valor x=8, entonces el arreglo después de ser ordenado por Quicksort podría quedar como [6,4,3,1,8,22,11,9], de esta manera el pivote x=8 queda acomodado en su posición y se generan dos pedazos a partir del original, los cuales a su vez deben de ser ordenados por Quicksort. Cuando se termina de ordenar cada pedazo se deben combinar para así obtener la solución a la entrada original.

Como se puede ver, lo primero que hace Quicksort es ordenar los elementos de forma que elementos menores que x quedan a la izquierda y elementos mayores quedan a la derecha, realizando así una partición del arreglo, en donde el pivote x suele tomarse como el primer elemento, es decir x=A[primero]. El algoritmo de la partición asume que el elemento A[último] es por lo menos tan grande como A[primero], de lo contrario se considera que el arreglo o la lista se encuentran vacíos. Si la entrada es de tamaño n, entonces inicialmente Quicksort deberá llamarse con primero = 0 y último = n - 1 pues el arreglo o la lista con la

que se esté trabajando debe revisarse de principio a fin para llevar a cabo la primer Partición. [1]

De forma muy general, el algoritmo quedaría definido como sigue: [3]

Algoritmo: Quicksort recursivo

Entrada: Un arreglo de n elementos: A[primero: último]

Salida: A[primero : último] ordenado en un orden no decreciente

```
if primero < último
        Partición(A[ 0 : n-1 ], primero, último, posición)
        Quicksort(A[ 0 : n-1 ], primero, posición-1)
        Quicksort(A[ 0 : n-1 ], posición+1, último)
endif
```

La parte más complicada del algoritmo (comparaciones, cambiar elementos de posición) se sitúa en "Partición". Una ventaja en la forma descrita por Hoare para llevar a cabo el reordenamiento es que los elementos se pueden mover muchas posiciones después de una sola comparación.

En "Partición" lo que hacemos es tomar el primer elemento del arreglo $A[primero]$ como pivote y extraerlo dejando así un espacio en el arreglo original. Una vez que tenemos este pivote, podemos comenzar a buscar elementos de derecha a izquierda, es decir desde $A[último]$ en dirección a $A[primero]$, para encontrar elementos pequeños, o en otras palabras elementos menores que nuestro pivote, de tal manera que cuando nos encontremos con uno que caiga dentro de la condición $A[índice] < pivote$ lo podamos mover hacia el espacio que quedó libre, recorriendo así más de una posición de ser necesario y permitiéndonos reacomodar el arreglo dejando del lado izquierdo elementos menores que pivote (en el algoritmo de Partición estas operaciones serán las que realice el método MoverChicos). Al mover este elemento quedará libre un espacio en su lugar, de tal manera que ahora tiene que buscarse un elemento grande, es decir un elemento mayor que pivote, partiendo de $primero + 1$ en dirección a $último$, cuando encontremos un elemento que cumpla la condición $A[índice] > pivote$, tendrá que ser movido las posiciones que requiera para colocarse en el espacio libre (estas operaciones serán las que realice el método MoverGrandes), dejando un nuevo espacio vacío en el arreglo. Este proceso se repite n veces, para no repetir ordenamientos se debe llevar el control de un par de índices, los cuales nos servirán para saber en que posición se encuentran los espacios libres para elementos pequeños y para elementos grandes, mientras esto siga ocurriendo llegará un punto en el cual los dos índices se encuentren, este será el punto en el cual termina la ejecución de "Partición".

El algoritmo para Partición quedaría definido como sigue: [3]

Algoritmo: Partición, Quicksort

Entrada: Un arreglo de n elementos A[primero : último]

Salida: Índice del arreglo, tal que los valores desde primero hasta índice - 1 son menores que el valor de pivote, y los valores desde índice + 1 hasta último son mayores.

Comentario: El espacio que quedará vacío al mover elementos chicos es espacioParaGrandes, el espacio que quedará vacío al mover elementos grandes es espacioParaChicos.

```
pivote x ← A[ primero]
índice para elementos chicos IC ← primero
índice para elementos grandes IG ← último
while IC < IG do
        espacioParaGrandes  ← MoverChicos(A, pivote, IC, IG)
        espacioParaChicos ← MoverGrandes(A, pivote, IC + 1,espacioParaGrandes)
        IC ← espacioParaChicos
        IG ← espacioParaGrandes – 1
endwhile
return IC
```

Nota: A espacioParaGrandes se le resta 1 porque al final del ciclo el espacio al cual hace referencia espacioParaGrandes ya ha sido ocupado mediante el método MoverGrandes, para evitar que esa posición del arreglo vuelva a ser procesado, movemos IG una posición hacia la izquierda.

Para terminar de entender el funcionamiento del algoritmo para la Partición utilizaremos el ejemplo de la Figura 4.1 con el arreglo de enteros [8,6,3,4,22,9,1,11] y mostraremos paso a paso como se reordena:

[8, 6, 3, 4, 22, 9, 1, 11] ← Entrada original

Se toma el 8 como pivote, se extrae del arreglo y se guarda en un espacio temporal

↓

[___, 6, 3, 4, 22, 9, 1, 11] ← Se crea un espacio vacío en donde se extrae el pivote

índice IC índice IG

↓ ↓

[___, 6, 3, 4, 22, 9, 1, 11] ← Se crean los índices para control de "Partición"

[1, 6, 3, 4, 22, 9, ___, 11] ← Queda un espacio vacío en donde se encontraba 1

[1, 6, 3, 4, ___, 9, 22, 11] ← Se reacomoda el 22 en el espacio vacío

[1, 6, 3, 4, _8_, 9, 22, 11] ← Como todos los elementos menores que x= 8 quedaron del lado izquierdo y los mayores del lado derecho, se coloca el pivote en el espacio vacío, dejándolo de esta forma en su posición correcta.

Lo que sigue es procesar el lado izquierdo y el lado derecho en referencia a el primer pivote que se eligió (x=8).

Lado izquierdo:

[1, 6, 3, 4] ← Nueva entrada temporal

[___, 6, 3, 4] ← Se toma como pivote el 1, dejando en su lugar un espacio vacío

[_1_, 6, 3, 4] ← Como a su derecha no hay valores menores, significa que el pivote ya se encontraba en su posición por lo que se vuelve a colocar; se debe procesar los elementos restantes.

[6, 3, 4] ← Nueva entrada

[___, 3, 4] ← Pivote x=6

[4, 3, ___] ← Se mueve el 4

[4, 3, _6_] ← Como ya no hay más elementos que mover se coloca el pivote en su posición.

[4, 3] ← Nueva entrada

[___, 3] ← x=4

[3, _4_] ← Se coloca el pivote en su posición

De esta manera la parte izquierda del arreglo queda ordenada de la siguiente manera:

[1, 3, 4, 6, 8]

Aún falta procesar el lado derecho respecto a x=8.

[9, 22, 11] ← Nueva entrada

[___, 22, 11] ← x = 9

[_9_, 22, 11] ← El pivote se coloca en su posición pues no hay elementos menores a la derecha

[22, 11] ← Nueva entrada

[___, 11] ← x=22

[11, ___] ← El 11 es menor que el pivote por lo tanto se mueve al espacio libre dejando uno en su lugar

[11, _22_] ← Se coloca el pivote en su posición

Finalmente el arreglo completo ha sido ordenado, quedando como salida:

[1, 3, 4, 6, 8, 9, 11, 22] ← Arreglo ordenado

Figura 4.1 – Partición, Quicksort

4.1.3 Análisis de peor caso: Quicksort

Sabemos que Quicksort realiza una Partición de la entrada original con lo cual va generando subintervalos de valores para analizarlos por separado.

Si suponemos que el arreglo es de tamaño n, y sabemos que el pivote que se toma siempre es el primer elemento, y que el resto de los elementos deben ser comparados con ese pivote, tenemos que habrá n – 1 comparaciones en el algoritmo de la Partición.

Por lo tanto el peor caso que podemos encontrar en Quicksort es en el que se tenga que comparar todos los elementos y no se muevan de posición, es decir el caso en el que el arreglo de entrada ya se encuentre ordenado. Cuando el arreglo se encuentre ordenado lo único que podrá hacer el algoritmo Partición es obtener subintervalos de n - 1 y dejar el pivote siempre en su lugar. De tal manera cada que se invoque Partición tendrá que hacer n-1 comparaciones, en donde n irá variando respecto al tamaño de cada subintervalo.

Por lo tanto obtenemos que:

$$W(n)=\sum_{k=2}^{n} n-1$$

Por lo tanto:

$$W(n)=\frac{n(n-1)}{2}$$

$$W(n)=n^2 \in O(n^2)$$

4.1.4 Análisis de recursividad [5]

Como vimos, Quicksort es un algoritmo que emplea recursividad, es decir tiene partes como la Partición, que emplean su propia definición para llevar a cabo el proceso de separación de la entrada original.

Para poder analizar este tipo de algoritmos en muchas ocasiones es necesario contar con una herramienta adicional, conocida como "ecuaciones de recurrencia", en el caso particular de Quicksort, esta ecuación nos será útil para el análisis de complejidad promedio.

Las ecuaciones de recurrencia se definen de manera muy sencilla. Se definen como una función sobre el conjunto de los números naturales, es decir $f(n):N$, en donde n nos servirá para especificar de cierta forma el tamaño del problema con el que nos estemos enfrentando. Las ecuaciones de recurrencia nos servirán para poder medir la cantidad de trabajo que realiza un algoritmo en particular, como se menciona en el capitulo anterior, lo que mejor nos funciona como elemento a medir son el número de operaciones básicas que realiza.

El miembro derecho de las ecuaciones de recurrencia $f(n)$ que podamos generar, se formará mediante la estimación de los costos de los bloques de los cuales esté formado nuestro algoritmo recursivo. Para poder llevar a cabo esta estimación debemos ponernos de acuerdo en algunas cosas:

1. Podremos dividir nuestro algoritmo en "bloques", por ejemplo una estructura de decisión *if..else* podrá formar 2 bloques. O podemos considerar cada renglón escrito en nuestro código como un bloque.
2. Cuando haya más de 1 bloque, deberemos sumar sus costos.
3. Cuando existan estructuras de decisión deberemos utilizar el costo máximo de las alternativas.
4. Si existen invocaciones no recursivas a otros métodos deberemos estimar el costo de dichos métodos.
5. Si existen invocaciones recursivas deberemos determinar el tamaño de su parámetro en función de n, es decir $n'(n)$. Por lo que obtendremos la función $f(n'(n))$.
6. El costo de las ecuaciones estará compuesto de los costos recursivos (aquellos términos del lado derecho que contienen a f) y no recursivos (aquellos términos del lado derecho que no contienen a f).

Para comprender como se define la ecuación de recurrencia se utilizará un ejemplo sencillo que se ha trabajado desde el capítulo anterior, sin embargo en esta ocasión con recursividad:

Algoritmo: Búsqueda de forma secuencial con recursividad

Entrada: arreglo A de enteros, un índice de búsqueda, tamaño del arreglo, elementoBuscado

Salida: Verdadero si se encuentra el elemento, falso si no se encuentra.

```
if índice de búsqueda >= tamaño
   resultado ← Falso
else
   if A[índice de búsqueda] == elementoBuscado
     resultado ← Verdadero
   else
     resultado ← busquedaSecuencialRecursiva(A, índice de búsqueda + 1,tamaño,
                  elementoBuscado)
return resultado
```

Como puede verse este algoritmo hace la búsqueda de la misma manera que el algoritmo presentado en la sección 3.2.5 pero elimina el uso de ciclos y en su lugar emplea recursión.

Para la definición de la ecuación de recurrencia el tamaño del problema estará definido como el número de elementos que contiene el arreglo menos el índice de búsqueda. Como el método se llamará de forma recursiva es obvio que el intervalo del arreglo en el cual se podrá presentar el elemento buscado se irá recortando debido a que en cada llamada de busquedaSecuencialRecursiva el índice de búsqueda se incrementa en 1. Por lo tanto n = *tamaño – índice de búsqueda.*

Para poder llevar a cabo la estimación del costo, supondremos que cada renglón del código anterior es un bloque.

Las líneas en las cuales existe un else se consideran como alternativas, una vez que se han leído las líneas de alternativas el algoritmo regresa el resultado, por lo tanto el costo de la última línea debe sumarse al de las alternativas.

El análisis de recursividad requiere de un caso base, el cual podremos excluir del análisis, además nuestro análisis será válido únicamente para n mayores que dicho caso base, en el algoritmo de búsqueda secuencial recursiva nuestro caso base se encuentra en la línea dentro del primer if, que es el caso en el que no se realiza comparación alguna con el arreglo, por lo tanto el costo de esta línea se excluye. Respecto a las demás líneas del código, únicamente la línea que se encuentra dentro del segundo else es de tipo recursiva, por lo que las demás se trataran como enunciados simples y usaremos las comparaciones con el arreglo como elemento de medición; por lo que de los enunciados simples

únicamente la línea que contiene el if dentro del primer else tendrá un costo, y su costo será de 1 (valor constante por haber una comparación con un elemento del arreglo).

Para el análisis de la línea de código con recursividad, se tiene que tomar en cuenta que la invocación es con el índice de búsqueda incrementado, por lo que nuestra n debe cambiar respecto a la que se considero bajo las condiciones iniciales, de esta manera obtenemos que:

Si sabíamos que

n = tamaño – índice de búsqueda

Y bajo la invocación recursiva vemos que el lado derecho es en realidad:

tamaño – (índice de búsqueda + 1) = tamaño – índice de búsqueda – 1

Por lo tanto tenemos que también se debe restar 1 del lado izquierdo.

n – 1 = tamaño – índice de búsqueda – 1

De esta manera podemos concluir que el costo de la llamada recursiva será n-1, es decir tendremos $f(n-1)$.

Finalmente tenemos que obtener el costo total mediante la elección del mayor costo entre las alternativas y sumándole el costo de la última línea:

Tenemos los siguientes costos basándonos en el criterio establecido:

Línea	Costo
1	0
2	0
3	else - alternativa
4	1
5	0
6	else – alternativa
7	n-1
8	0

Figura 4.2 – Costos de cada bloque

Existen alternativas entre la línea 2 y 4, así como entre la linea 5 y 7, las alternativas (else) no tienen costo alguno. En la línea 2 se encuentra el caso base, no lo tomaremos en cuenta como una alternativa.

Por lo tanto obtenemos:

$(0 + (1 + Máximo(0, f(n-1))) + 0$

En donde Máximo debe elegir el costo mayor de entre las alternativas.

Entonces:

$$(0 + (1 + Máximo(0, f(n-1))) + 0 = 1 + f(n-1)$$

Finalmente obtenemos que la ecuación de recurrencia para este algoritmo es:

$$f(n) = f(n-1) + 1$$

Y el costo del caso base para este ejemplo y bajo el criterio de medición establecido será:

$$f(0) = 0$$

En caso de que n = 0, sabemos que desde el principio el caso base se cumplirá y el número de comparaciones será nulo, por lo tanto la ecuación de recurrencia funciona para n > 0.

Los problemas del tipo divide y vencerás tienen en común que se dividen en varios subproblemas, además es claro que cada uno de estos subproblemas es de un tamaño menor a n que es el tamaño original, por lo tanto tenemos X subproblemas de tamaño n/Y, quedando $Xf\left(\frac{n}{Y}\right)$, además como en casi todos los problemas de recursión habrá por lo menos un costo no recursivo que sumar, en el caso de divide y vencerás ese costo puede estar en la Partición o en la Combinación, por lo que de manera general podemos definir una ecuación de recurrencia para los problemas del tipo divide y vencerás de la siguiente manera:

$$f(n) = Xf\left(\frac{n}{Y}\right) + g(n)$$

en donde $g(n)$ es el costo no recursivo del algoritmo.

Una vez que hemos obtenido una ecuación de recurrencia debemos utilizar una herramienta adicional, los árboles de recursión, los cuales nos servirán para finalmente analizar el costo.

Para explicar mejor el desarrollo del árbol de recursión emplearemos el mismo ejemplo del algoritmo de búsqueda secuencial recursiva.

Los nodos de nuestro árbol estarán formados por dos partes, una de ellas será el tamaño expresado mediante la f que denota recursividad con una n que nos indica el tamaño y una segunda parte que será el costo no recursivo de la invocación, de tal manera que cada nodo se verá como se muestra en la siguiente figura:

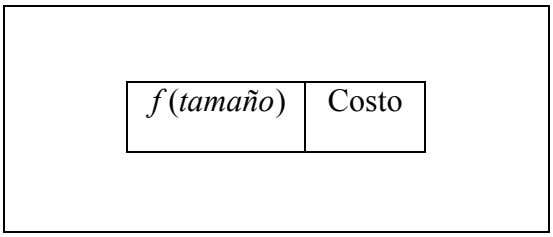
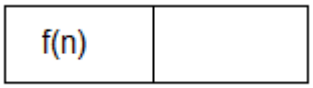


Figura 4.3 – Nodo del árbol de recursión

Recordando la ecuación de recurrencia obtenida anteriormente:

$$f(n) = f(n-1) + 1$$

Podemos identificar un primer nodo:

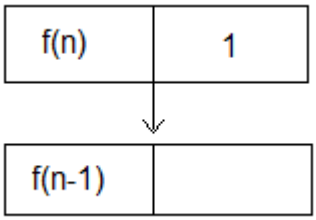


Este será el nodo raíz para nuestro árbol de recursión, hasta este punto se considera un nodo incompleto pues no tiene el costo no recursivo.

Para determinar el costo no recursivo y los hijos del nodo debemos evaluar nuestra ecuación de recurrencia en el tamaño del nodo actual y analizar la parte derecha de la ecuación.

Para la primer evaluación la ecuación de recurrencia queda igual. Los hijos del nodo serán aquellos factores del lado derecho que tengan a f y el costo no recursivo del nodo que se esta examinando serán los factores no recursivos.

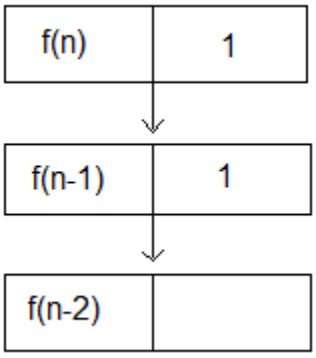
Con lo cual podemos obtener el costo del nodo actual, y un nodo hijo:



Para continuar el proceso, debemos primero evaluar la ecuación de recurrencia con el valor del tamaño de nuestro nuevo nodo actual, $f(n-1)$, con lo cual obtenemos:

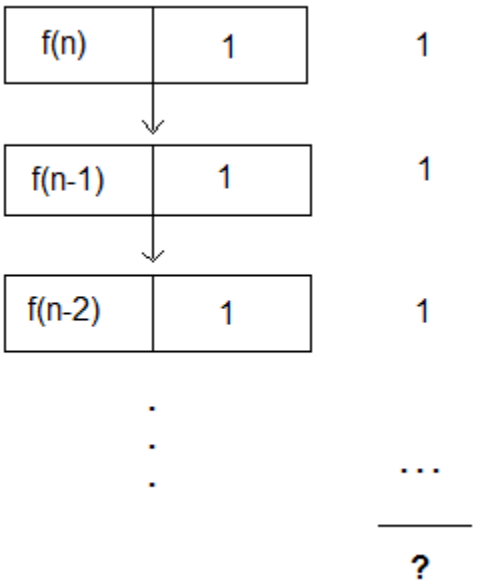
$$f(n-1) = f((n-1)-1) + 1 = f(n-2) + 1$$

Por lo tanto podemos obtener el costo no recursivo del nodo actual además del nodo hijo:



Podríamos continuar así y seguiríamos obteniendo nodos y costos no recursivos, por lo tanto podemos ver que mientras el árbol va creciendo su factor de tamaño se va modificando con un cierto patrón, podemos decir que a una cierta profundidad D, el tamaño es $n - D$, sabemos que la profundidad del nodo raíz es 0.

Una vez que tenemos esto, debemos evaluar el árbol de recursión, para ello debemos sumar los costos (lado no recursivo) de cada fila de nodos, en este ejemplo cada fila esta compuesta únicamente de un nodo, para este ejemplo obtenemos que:



Como puede verse, una vez que se obtiene la suma por fila debemos sumarlas todas.

Podemos deducir que si el tamaño del nodo es de aproximadamente n-D, el caso base se presentará cuando tengamos la profundidad $D = n$, esto puede verse claramente si se revisa el algoritmo (cuando se ha evaluado todo el arreglo y no se encuentra el valor buscado), además podemos ver que el costo no recursivo de todas las filas de nodos será 1.

De esta manera el total del árbol será:

$$(1)(n) = n$$

4.1.5 Análisis de complejidad promedio: Quicksort

Quicksort lleva este nombre por su comportamiento promedio que es mucho mejor que el que encontramos para el peor caso, puede consultarse la Figura 3.10 para ver la comparación entre ambas complejidades.

Para el análisis promedio debemos suponer primero que todos los elementos contenidos en el conjunto a ordenar son distintos.

Como se vio en el análisis de peor caso, sabemos que el algoritmo de Partición hará n – 1 comparaciones para entradas de tamaño n, de tal manera que al final de la ejecución de Partición habrá dos subintervalos, uno de tamaño i y otro de tamaño n – 1 – i que es el número total de elementos menos el tamaño del primer intervalo.

En el capitulo 3 se mencionó que para el análisis de comportamiento promedio es necesario tomar en cuenta un valor de probabilidad P(I) para cuando se presenta un cierto elemento I, para este análisis supondremos que I será el punto de partición, es decir el punto en el cual serán separados los dos intervalos después de Partición, y diremos que todos los puntos dentro del conjunto a ordenar tienen la misma probabilidad, es decir tienen probabilidad $P(I) = \frac{1}{n}$ en donde n es el tamaño de la entrada, de esta manera obtenemos la ecuación de recurrencia que estamos buscando:

$$f(n) = \left(\sum_{i=0}^{n-1} \frac{1}{n} (f(i) + f(n-1-i)) \right) + (n-1)$$

Como puede verse la parte dentro de la sumatoria corresponde a los costos recursivos y el $(n-1)$ que se suma al final es la parte del costo no recursivo de la ecuación, es decir la llamada al método Partición. $f(i)$ corresponde al tamaño de la primer llamada recursiva que hace el algoritmo y $f(n-1-i)$ a la segunda llamada, es decir el ordenamiento del primero y segundo intervalo respectivamente.

Los casos base se presentan cuando el tamaño n es 0 o 1, que es cuando no queda nada que ordenar, por lo tanto:

$$f(0) = f(1) = 0$$

Por lo que la ecuación de recurrencia es válida únicamente para $n \geq 2$.

Sabiendo esto podemos reducir un poco la ecuación de recurrencia, si sabemos que la segunda llamada recursiva $f(n-1-i)$ analiza el segundo intervalo, podemos deducir que ese intervalo irá de n – 1, hasta n – 1 = 0, por lo tanto irá de $f(n-1)$ hasta $f(0) = f(1)$, esto es exactamente igual a la sumatoria de 0 hasta n – 1 de la primer llamada recursiva $f(i)$, por lo tanto podemos decir que se esta evaluando dos veces la misma sumatoria, de esta forma obtenemos que:

$$f(n) = \left(\sum_{i=1}^{n-1} \frac{2}{n} f(i) \right) + (n-1) = \left(\frac{2}{n} \sum_{i=1}^{n-1} f(i) \right) + (n-1)$$

Para poder analizar el costo del caso promedio, debemos hacer una suposición, por ejemplo que después de la ejecución de Partición, los dos subintervalos que se obtienen son del mismo tamaño, por lo tanto si la entrada es de tamaño n, tendremos dos subintervalos de tamaño $\frac{n}{2}$. Si tomamos como elemento de medición operaciones básicas, y en específico las comparaciones que se llevarán acabo entre el primer y último elemento (ver algoritmo de Quicksort), y si observamos la ecuación de recurrencia general para los algoritmos del tipo divide y vencerás al cual pertenece Quicksort, podemos deducir lo siguiente:

Sabemos por la ecuación de recurrencia obtenida anteriormente que deberá tener un costo no recursivo el cual podemos expresar simplemente como n que es el tamaño de la entrada; además como ya vimos, habrá 2 llamadas recursivas y cada una será de tamaño $\frac{n}{2}$ que es el tamaño de cada subintervalo, de esta manera obtenemos la siguiente ecuación de recurrencia:

$$f(n) \approx 2f\left(\frac{n}{2}\right) + n$$

Para el diseño del árbol de recursión nos conviene por claridad para el lector separar las dos llamadas recursivas:

$$f(n) \approx f\left(\frac{n}{2}\right) + f\left(\frac{n}{2}\right) + n$$

Con esta ecuación y siguiendo la lógica explicada antes para desarrollar los árboles de recursividad obtenemos:

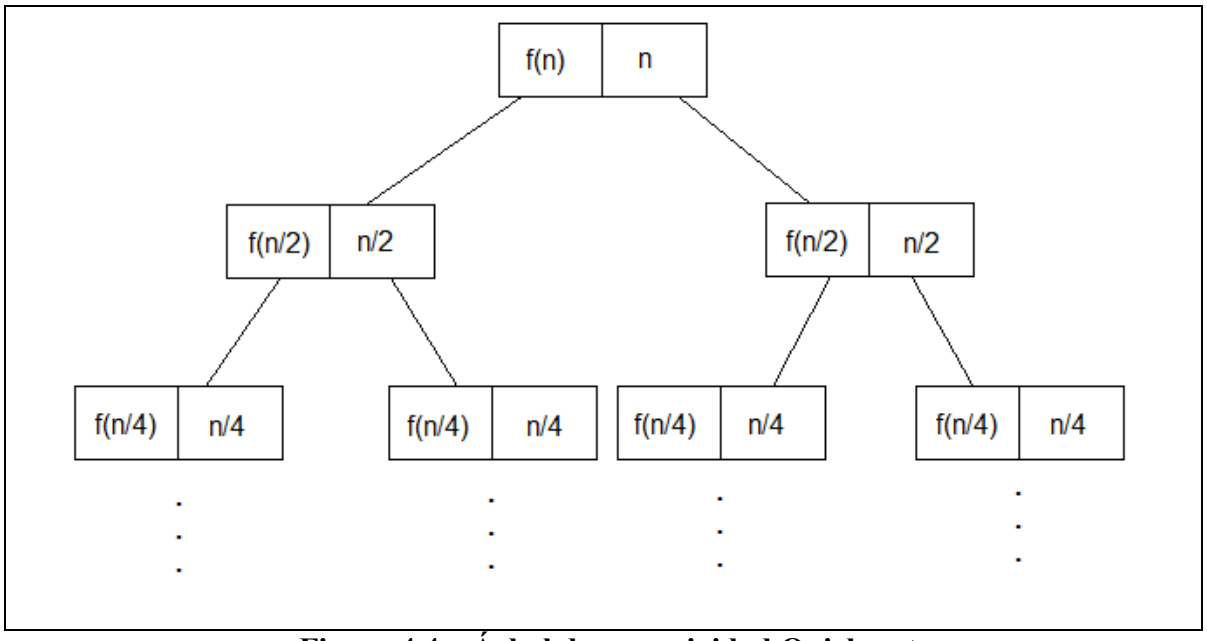


Figura 4.4 – Árbol de recursividad Quicksort

Podemos observar el patrón con el cual van modificándose los costos no recursivos del árbol, de tal manera que a una profundidad D, el costo será $\frac{n}{2^D}$, si despejamos D para obtener la profundidad a la cual se presentará el caso base obtenemos que $D = \log_2(n)$.

Podemos observar que la suma de cada fila de nodos es $\frac{n}{2}+\frac{n}{2}=\frac{n}{4}+\frac{n}{4}+\frac{n}{4}+\frac{n}{4}=n$, por lo tanto el costo total del árbol sería:

$$n\log_2 n \in O(n\log_2 n)$$

Lo cual es mucho mejor que el comportamiento en el peor caso.

4.2 Programación dinámica

4.2.1 Introducción [11]

La estrategia de programación dinámica obtiene la solución a problemas complejos o grandes, a partir de utilizar la solución de instancias chicas o más simples del problema. Las soluciones a los problemas más chicos o simples son construidas a su vez combinando la solución de problemas o instancias aún más chicas del problema. Por lo tanto comenzamos con la solución a los problemas más chicos o simples y vamos combinándolas para obtener la solución total.

La estrategia de programación dinámica parece de cierta manera muy parecida a la de Divide y vencerás por el hecho de que también ocupa divisiones para generar segmentos más chicos, sin embargo la diferencia principal es que la programación dinámica utiliza la solución de los problemas chicos para obtener la solución de los grandes, por lo tanto la programación dinámica generalmente no recalcula soluciones. De esta manera mejora por mucho la complejidad de los algoritmos.

Para solucionar un problema mediante la estrategia de Programación dinámica, debemos poder:

- Definir una solución como sucesión de decisiones.
- Definir una solución recursiva.
- Calcular el valor de la solución óptima al problema actual utilizando las soluciones parciales de los subproblemas previamente analizados.
- Obtener la solución total al problema utilizando los valores almacenados.

Como vimos tanto Divide y vencerás como la Programación dinámica requieren de la creación de subproblemas a partir de un problema principal. Pero ¿cuándo debemos elegir la estrategia de Programación dinámica? Generalmente cuando obtener la solución requiera de resolver muchas veces un mismo subproblema, ya que como vimos en los puntos anteriores esta estrategia guarda un registro de los cálculos previos para reutilizarlos.

4.2.2 Fibonacci [2]

Para poder entender la estrategia de Programación dinámica, uno de los ejemplos más sencillos es el problema del cálculo de la sucesión de números de Fibonacci.

La secuencia de Fibonacci se puede obtener mediante la función recursiva:

$$F(n) = \begin{cases} 1 & n \leq 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

Por lo tanto la sucesión de números esta compuesta por:

n	F(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13
7	21
8	34

9	55
10	89
11	144
12	233
13	377
14	610
.	.
.	.
.	.
.	.
.	.

Figura 4.5 – Sucesión de números de Fibonacci

Como puede verse comienza con 1 y luego cada número es la suma de los dos anteriores.

El algoritmo que mejor describe el comportamiento de esta función sería:

Algoritmo: Fibonacci recursivo

Entrada: Número entero n

Salida: Número de Fibonacci

```
if n <= 1
        return 1
else
        return Fibonacci( n – 1 ) + Fibonacci( n – 2 )
endif
```

La complejidad de este algoritmo es exponencial ya que gran parte del trabajo se repite con cada llamada recursiva, es decir en este algoritmo no se utilizan los cálculos previos y sabemos que a mayor profundidad del árbol recursivo habrá más nodos y por lo tanto mucho más cálculos que realizar para obtener el valor.

Sin embargo es bastante sencillo realizar un algoritmo que mantenga un registro de los valores previamente calculados, una opción es utilizar un arreglo, de esta manera cada vez que se haga una llamada no será necesario recalcular todos los valores, la solución puede darse mediante el siguiente algoritmo iterativo:

Algoritmo: Fibonacci iterativo

Entrada: Arreglo para almacenar valores calculados, número entero n

Salida: Número de Fibonacci

```
if n<=1
        return 1
else
        Arreglo[ 0 ] ← 1
        Arreglo[ 1 ] ← 1
        for i ← 2 to n do
                Arreglo[ i ] ← Arreglo[ i – 1 ] + Arreglo [ i – 2 ]
        endfor
        return Arreglo[ n ]
endif
```

Este algoritmo ya no requiere de recalcular todos los valores cada vez que se llega a un valor de n mayor puesto que puede utilizar un registro de los valores previamente calculados; de esta manera si se toma como operación básica la asignación que se hace al arreglo dentro del ciclo, la complejidad del algoritmo que se obtiene es lineal, es decir $O(n)$.

De esta manera queda claro que en los casos que sea posible, es decir siempre y cuando los problemas se superpongan, podemos recurrir a un algoritmo que mantenga un registro y con eso obtendremos una mejora considerable en la complejidad. Esta es una de las principales ventajas de la Programación dinámica frente a la estrategia de Divide y vencerás.

4.2.3 Dijkstra

Este algoritmo nos sirve para encontrar el camino mínimo dado un vértice origen hacia el resto de los vértices dentro de un digrafo con pesos positivos en cada arista, es decir un grafo que tiene dirección entre cada uno de sus vértices.

El algoritmo de Dijkstra funciona probando todos los vértices desde el origen a los demás vértices buscando los caminos más cortos, una vez que se han calculado todos los caminos y por lo tanto visitado todos los vértices el algoritmo se detiene.

Para poder obtener la solución a este algoritmo mediante la estrategia de Programación dinámica, primero debemos asegurarnos de que cumpla con el principio óptimo, el cual dice que dado un problema de optimización, si la solución total al problema es óptima entonces cada subconjunto que compone la solución total también será óptimo. En el caso de este algoritmo esto nos puede servir para asegurar que dado un vértice origen v y otro vértice cualquiera j del cual se tiene el camino más corto, podemos asegurar que si entre estos dos vértices se encuentra otro vértice k, entonces los caminos entre v y k, y entre k y j, será también mínimos.

El algoritmo que podemos obtener bajo esta estrategia queda como sigue: [3]

Algoritmo: Dijkstra

Entrada: Arreglo bidimensional con los pesos de todas las aristas, número de nodos, arreglo unidimensional en donde se almacenarán los caminos mínimos

Salida: Camino mínimo del nodo origen a todos los demás

```
1   for i ← 2 to n do
2           Elementos visitados [ i ] ← falso
3           Camino mínimo [ i ] ← Arco [ 1, i ]
4   endfor
5   Elementos visitados [ 1 ] ← verdadero
6   posición ← 1
7   for i ← 2 to n-1 do
8           posición ← Mínima Distancia(Camino mínimo, Elementos visitados, posición)
9           Elementos visitados [ posición ] ← verdadero
10          for j ← 2 to n do
11                  if Elementos visitados [ j ] != verdadero
12                          Camino mínimo [ j ] ← Mínimo(Camino mínimo[ j ], Camino mínimo[ posición] +
13                                                  Arco[ posición, j ]
14                  endif
15          endfor
16  endfor
```

Para este algoritmo es necesario numerar las líneas para poder explicarlas.

El ciclo que se encuentra entre las líneas 1 y 4 sirve para inicializar; pone todos los nodos en estado falso, lo cual servirá para identificar cuales nodos han sido visitados y de esa manera no repetir cálculos, que es uno de los objetivos principales de la programación dinámica. Arco contiene los pesos entre nodos, si no existe conexión entre nodos contiene un valor lo suficientemente grande (∞) para identificar que no hay conexión. La línea 3, inicializa Camino mínimo con todos los pesos desde el vértice origen.

La línea 5 identifica el primer nodo como un nodo visitado, de tal manera que no se considere a si mismo para el análisis de camino mínimo.

En el ciclo de la línea 7 es donde el algoritmo llevará a cabo la mayor parte de sus cálculos, por lo tanto este ciclo y el ciclo interior son los que debemos considerar para obtener la complejidad del algoritmo.

En la línea 8, se hace una llamada al método Mínima Distancia, el cual obtiene la posición del siguiente vértice en el que se encuentra la siguiente mínima distancia, el algoritmo para este método es el que sigue:

Algoritmo: Mínima Distancia, se utiliza en Dijkstra

Entrada: Arreglo unidimensional que contiene los caminos mínimos, el arreglo de elementos visitados y la posición actual.

Salida: Posición del siguiente vértice cuya arista es de menor peso

```
menor distancia ← Número lo suficientemente grande (∞)
posición ← 1
for i ← 2 to n do
        if Elementos visitados [ i ] != verdadero
                if Camino mínimo [ i ] < menor distancia
                        menor distancia ← Camino mínimo [ i ]
                        posición ← i
                endif
        endif
endfor
return posición
```

El algoritmo de Mínima Distancia es bastante sencillo, simplemente busca la siguiente menor distancia en el conjunto de todos los nodos que no han sido visitados y respecto al arreglo de caminos mínimos ya calculado.

La línea 9 del algoritmo marca el siguiente vértice al cual se debe mover como visitado.

La línea 10 comienza el ciclo interior, el cual busca únicamente dentro de los vértices no visitados. El método al cual se hace invocación (Mínimo) sirve únicamente para obtener el valor mínimo entre dos números, en este caso compara para saber si el valor de peso ya calculado es el menor, o si existe un menor camino entre el peso en la posición del vértice actual más el peso del arco desde el vértice actual hacia el vértice siguiente j.

Es fácil hacer el análisis de complejidad de esta algoritmo, como se mencionó antes la mayoría del trabajo se realiza en los dos ciclos (línea 7 a las 16), si tomamos como elemento de medición el número de asignaciones, veremos que independientemente de las asignaciones que se hagan adentro de los ciclos, en el primero se asignará n veces un valor a i y en el ciclo interno, se asignará n veces un valor a j, por lo que podemos concluir que la complejidad de Dijkstra será de $O(n*n) = O(n^2)$.

Como hemos visto, para poder decir que un algoritmo se puede considerar de Programación dinámica, debemos asegurar que cumple con el principio óptimo.

4.3 Algoritmos codiciosos

4.3.1 Introducción

Esta es una estrategia muy utilizada para problemas en los que se desea maximizar o minimizar, es decir algoritmos de optimización. Los algoritmos que se basan en la estrategia codiciosa generalmente son fáciles de programar, y por lo regular son eficientes. [5]

La estrategia de Algoritmos codiciosos radica en tomar una sucesión de decisiones (por lo tanto se trabaja por etapas), asegurándonos de que cada decisión/etapa es por si misma la mejor decisión local posible, sin tomar en cuenta las futuras repercusiones de dicha decisión, es posible que más adelante nos demos cuenta de que alguna decisión no fue la mejor, sin embargo los algoritmos codiciosos no permiten deshacer o revertir. Esto quiere decir que mediante el planteamiento de una solución codiciosa no necesariamente llegaremos a una solución óptima. Antes de tomar una decisión el algoritmo codicioso debe comprobar si es una decisión capaz de llevarnos hacia la solución, en caso de que no sea capaz se descarta totalmente y se busca otra decisión. Cada vez que se concluye una etapa, el algoritmo comprueba si se ha obtenido una solución, en caso afirmativo termina su ejecución. [5]

Para poder aplicar esta estrategia sobre un problema, debemos considerar que si la estrategia se basa en una sucesión de mejores decisiones, por lo tanto deberemos ser capaces de desarrollar un método que nos permita hacer dicha tarea (SelecciónCodiciosa), además debemos poder obtener un método que compruebe si las decisiones tomadas son buenas de tal manera que nos permita seguir avanzando en la sucesión de decisiones y lleguemos a una solución, finalmente debemos poder obtener una función la cual queremos optimizar, es decir una función que nos determine el valor de nuestra solución.

Con todo lo anterior podemos escribir el algoritmo codicioso de forma general: [5]

Algoritmo: Algoritmo codicioso

Entrada: Un conjunto base S

Salida: Un subconjunto ordenado de S que optimiza la función objetivo f, o devuelve un mensaje de que no se pudo encontrar una solución

```
Solución parcial ← vacío
R ← S
while Solución parcial no sea una solución and  R ≠ vacío  do
        X ← SelecciónCodiciosa(R)
```

```
        R ← R\{x}
        if Solución parcial ∪ {x} es factible
                Solución parcial ← Solución parcial ∪ {x}
        endif
endwhile
if Solución parcial es una solución
        return Solución ← Solución parcial
endif
else
        return "El algoritmo codicioso no puede encontrar una solución"
endif
```

En muchas aplicaciones de este algoritmo, se proporciona un peso para los elementos de S, de tal forma que SelecciónCodiciosa simplemente escoge entre el mayor o menor elemento en R. Sin embargo, no siempre es la mejor decisión, y como se mencionó antes, por una mala selección en dicho método podemos terminar con una solución no óptima.

4.3.2 Algoritmo de Kruskal

Es un algoritmo que nos sirve para encontrar un árbol abarcante mínimo de grafos ponderados, conexos y no dirigidos. Lo que se pretende es por lo tanto generar un subconjunto de aristas a partir de una entrada original, las cuales tengan incluidas a todos los vértices del árbol y que además el valor total de todas las aristas seleccionadas sea el mínimo posible.

En el algoritmo de Kruskal se escoge en cada paso las aristas con menor peso de cualquier punto del grafo siempre sin tomar en cuenta las que formen un ciclo con cualquier arista que ya haya sido escogida. El algoritmo termina cuando tenemos todos los nodos del grafo conectado en alguna de las aristas escogidas, osea cuando tengamos n-1 arcos, si sabemos que n son el número de nodos de nuestro grafo.

En general, el algoritmo de Kruskal puede escribirse de la siguiente manera: [3]

Algoritmo: Kruskal

Entrada: Grafo ponderado G, no dirigido y conexo

Salida: Un subconjunto que forma un árbol abarcante mínimo de G

```
Ordenamos mediante cualquier algoritmo de ordenamiento los arcos del grafo de
acuerdo a su peso
Tamaño ← 0
for i ← 0 to n-1 do    //cada vértice en una partición distinta
        P[i] ← i
endfor
j ← 0
```

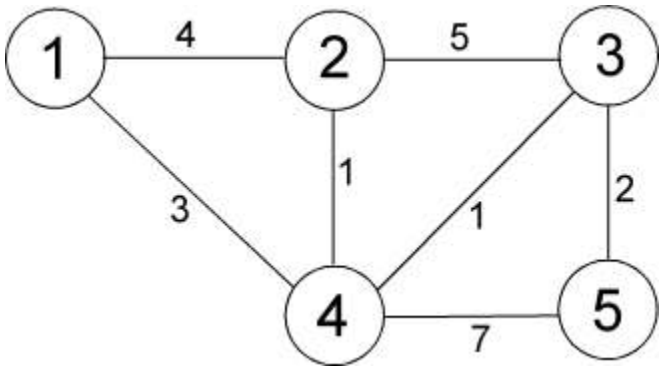
```
n ← número de vértices
while Tamaño ≤ n - 1 and índice < número de arcos
        Se incrementa el índice
        //arco e = uv, en donde u es el vértice origen y v el destino
        1er componente ← Obtener el valor de la partición a la que pertenece u
        2da componente ← Obtener el valor de la partición a la que pertenece v
        if 1er componente != 2da componente
                Unir(partición, 1er componente, 2da componente)
                Se incrementa tamaño
                Solución ← Solución ∪ {e}
        endif
endwhile
```

Como puede verse este algoritmo hace a lo mucho n-1 uniones, en donde n es el número de vértices, y a lo mucho debe encontrar m arcos, por lo tanto para estos procedimientos tenemos una complejidad asintótica O(n).

Finalmente podemos observar que el algoritmo considera que existe una ordenación al principio, la complejidad del ordenamiento puede ser de $O(m \log m)$ en donde m es el número de los arcos que tiene el grafo, esto es equivalente a $O(m \log n)$. Por lo tanto obtenemos que la complejidad del algoritmo de Kruskal es $O(m \log n)$.

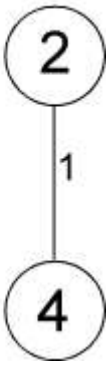
Un ejemplo gráfico de ejecución del algoritmo puede ayudar a comprenderlo:

Sea el siguiente grafo G conexo, no dirigido y ponderado:

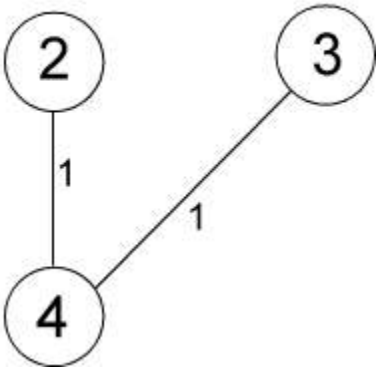


Como se mencionó en el algoritmo de Kruskal se van seleccionando las aristas de menor peso siempre y cuando estas no formen un ciclo con las ya escogidas, el algoritmo queda ejemplificado mediante los pasos siguientes:

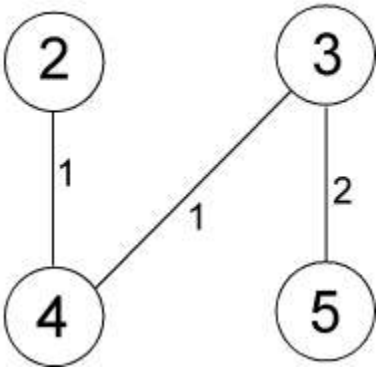
PASO 1:

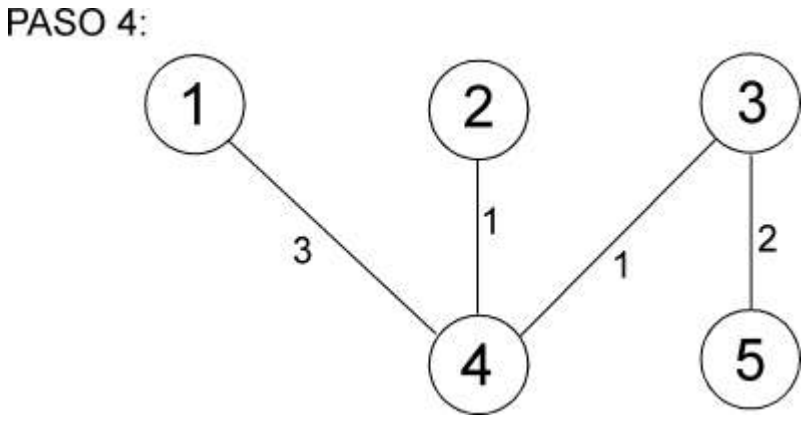


PASO 2:



PASO 3:





Con lo cual obtuvimos un subconjunto de aristas, vemos que se incluyen todos los vértices, y además el valor de las aristas en el árbol es el mínimo.

4.3.3 Algoritmo de Prim

El algoritmo ideado por Prim sirve para encontrar un árbol abarcante mínimo en un grafo no dirigido, conexo y ponderado, por lo tanto se trata de un algoritmo que tiene prácticamente las mismas condiciones que el algoritmo de Kruskal; sin embargo, como veremos trabaja de forma distinta:

Este algoritmo comienza seleccionando un vértice de forma arbitraria (se convertirá en el vértice inicial y se debe meter en la solución), luego debemos seleccionar la arista de menor peso conectada a un vértice seleccionado anteriormente, elegimos el siguiente vértice que no se encuentre incluido en la solución y que este conectado a esta arista, iteramos en este último paso mientras la arista elegida conecte a un nodo no seleccionado con un nodo seleccionado, el algoritmo termina su ejecución cuando se hayan seleccionados todos los nodos del grafo.

De forma general el algoritmo puede escribirse de la siguiente manera: [3]

Algoritmo: Algoritmo de Prim

Entrada: Grafo ponderado G, no dirigido y conexo, w que contiene todos los pesos de las aristas

Salida: Un subconjunto que forma un árbol abarcante mínimo de G

Debemos inicializar todos los vértices de tal manera que aparezcan como no seleccionados:

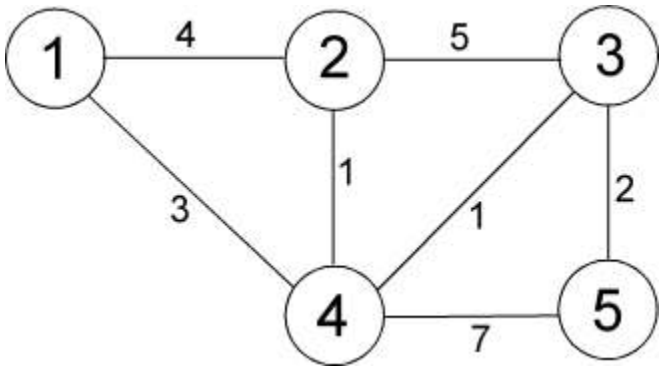
```
for v ← 0 to n – 1 do
        Distancia[v] ← ∞
        EnElÁrbol[v] ← falso
endfor
```

```
Padre[r] ← -1
Distancia[r] ← 0
for Etapa ← 1 to n – 1 do
        Seleccionar vértice u que minimice a Distancia[u] sobre todos los vértices u
que cumplan con que EnElÁrbol[u] = falso
        EnElÁrbol[u] ← verdadero
        for cada vértice v tal que uv ∈ A do // A es el conjunto de aristas
                if EnElÁrbol[v] = falso
                        if w(uv) < Distancia[v]
                                Distancia[v] ← w(uv)
                                Padre[v] ← u
                        endif
                endif
        endfor
endfor
```

Sabemos que el for exterior se va a ejecutar n – 1 veces en donde n es el número de vértices que tiene el grafo, y que el for anidado se ejecutará por lo menos una vez por cada vértice que se encuentre conectado a una arista, por lo que tenemos una complejidad $O(n^2)$.

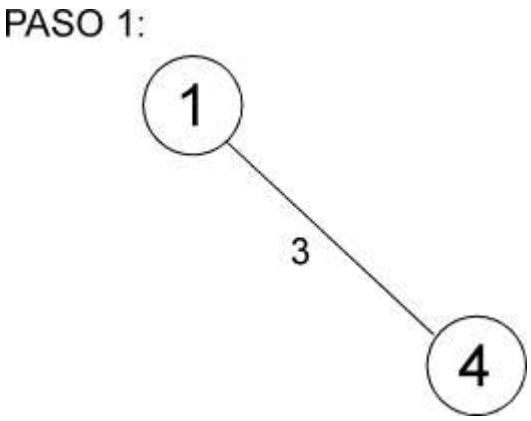
Al igual que hicimos antes, nos conviene para que quede claro el funcionamiento del algoritmo, ejemplificarlo gráficamente:

Utilizaremos el mismo grafo G que en el algoritmo anterior:

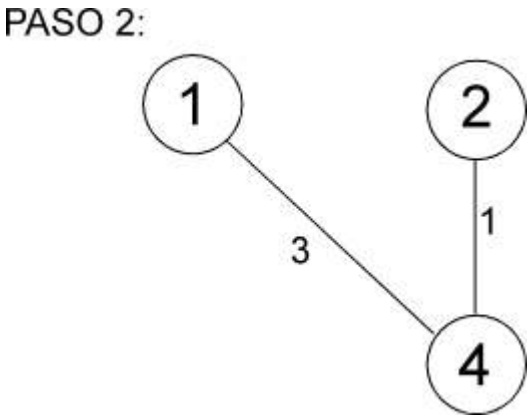


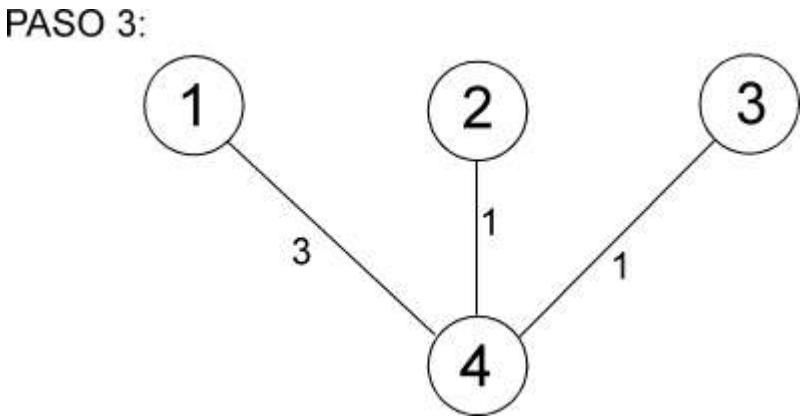
Seguiremos paso a paso el algoritmo hasta obtener el árbol mínimo abarcante para G:

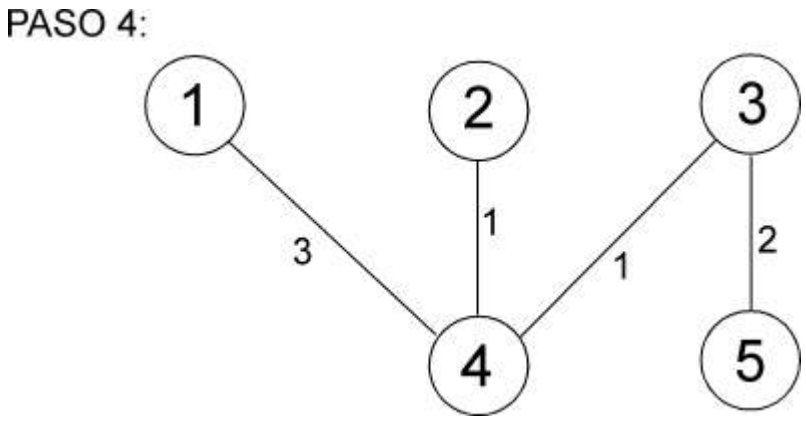
Elegimos arbitrariamente al vértice 1 como nuestro vértice inicial, el peso mínimo de los arcos conectados a este vértice se encuentra en la unión 1:4, por lo tanto lo incluimos en la solución:



En el árbol resultante debemos buscar el arco de menor peso que conecta a alguno de los vértices contenidos en la solución con alguno de los vértices que no han sido seleccionados, de esta manera obtenemos:







Observamos que ambos algoritmos resuelven el mismo problema y sin embargo cada uno tiene distinto orden de complejidad, ¿cuál debemos elegir? Dependerá del grafo con el que nos enfrentemos. El algoritmo de Kruskal depende tanto del número de arcos como de vértices. Por ejemplo en los grafos en que el número de arcos es cercano al número de vértices, la complejidad de Kruskal (en este caso sería aproximadamente $O(n \log n)$) será mejor que la de Prim $(O(n^2))$, en casos contrarios, posiblemente el algoritmo de Prim tenga una complejidad menor.

Ambos algoritmos son característicos de la estrategia de algoritmos codiciosos, pues como hemos visto siguen una lógica de ganar-ganar tomando en cada etapa lo mejor posible, sin tomar en consideración futuras repercusiones. Es lógico que en algunos casos una mejor decisión local pudiera llevar a que más adelante nos encontremos con decisiones que hagan que la solución final no sea óptima.

4.4 Algoritmos de retroceso

4.4.1 Introducción

Hasta ahora hemos visto varias estrategias de diseño, y hemos podido observar que existen ciertas condicionantes para poder utilizar una u otra, por ejemplo en el caso de la Programación dinámica vimos que ha de cumplirse el principio óptimo, en el caso de los Algoritmos codiciosos pudimos comprender que no podemos obtener una solución para todos los problemas.

Por lo tanto la estrategia de algoritmos de retroceso nos sirve como complemento a aquellos problemas que no pueden ser resueltos por las estrategias anteriores. Siempre y cuando podamos resolver el problema en etapas podremos utilizar la estrategia de retroceso para obtener todas sus posibles soluciones.

Esta estrategia prueba todas las posibilidades hasta encontrar la mejor; se puede ver como una búsqueda primero en profundidad de todas las posibles soluciones. Durante la búsqueda si una alternativa no funciona, se retrocede hasta una decisión tomada y se prueba otra alternativa. Si se acaban todas las alternativas, se retrocede aún más en las decisiones tomadas, y se prueba con otra alternativa. Si no podemos retroceder significará que el algoritmo ha fallado. Conforme vamos avanzando iremos obteniendo soluciones parciales del problema. [3]

En general los algoritmos de retroceso no funcionan bajo una regla bien establecida, simplemente se trata de ir probando en busca de una solución; el problema se presenta cuando las pruebas son demasiadas pues el algoritmo podría tardar demasiado en hallar una solución.

De esto podemos ver que la dificultad al implementar un algoritmo bajo esta estrategia se presentará en la definición de las restricciones que nos permitan definir si una alternativa es buena o no para la solución.

Un ejemplo clásico de los algoritmos de retroceso es el de las ocho reinas.

4.4.2 Problema de las ocho reinas

El problema radica en colocar 8 reinas en un tablero de ajedrez, de tal manera que ninguna reina este atacando a otra reina. Una reina puede moverse y por lo tanto atacar, según las reglas del ajedrez, de manera horizontal, vertical y diagonalmente. De esta manera sabemos que cada reina debe estar en un renglón diferente, columna diferente y diagonal diferente.

Para poder resolver el problema, de forma natural comenzaríamos colocando una reina en el tablero, en el primer renglón y primer columna, después colocaríamos otra reina sin que se pueda atacar con la primera, y seguiríamos colocando así una reina tras otra; ¿qué sucede si en algún momento no podemos colocar una reina sin que esta sea atacada por alguna otra ya colocada en el tablero? De nuevo, la solución natural sería recolocar la última reina que pusimos en el tablero de tal manera que deje un espacio para la que estamos tratando de poner, ¿y si eso no es suficiente? Tendríamos que mover la última y la penúltima, y así hasta encontrar un ordenamiento que nos permita encontrar la solución al problema. Este procedimiento de regresar para después volver a avanzar en busca de la posible solución, es lo que convierte al problema de las ocho reinas, en un ejemplo idóneo de los algoritmos de retroceso. Este proceso puede escribirse con el siguiente algoritmo:

Hasta ahora podemos decir, que el problema de ocho reinas si se puede resolver en etapas, en cada una de las cuales habremos de decidir en que posición colocar una reina. Para estas decisiones, deberemos tomar en cuenta algunas restricciones que nos indiquen la relación que debe haber entre todas las posibles posiciones que pueden formar parte de la solución, de lo contrario tendríamos demasiadas alternativas en cada etapa.

Para mostrar una posible forma de solucionar el problema y que quede claro que este algoritmo es muy parecido a una búsqueda primero en profundidad, acotaremos un poco el

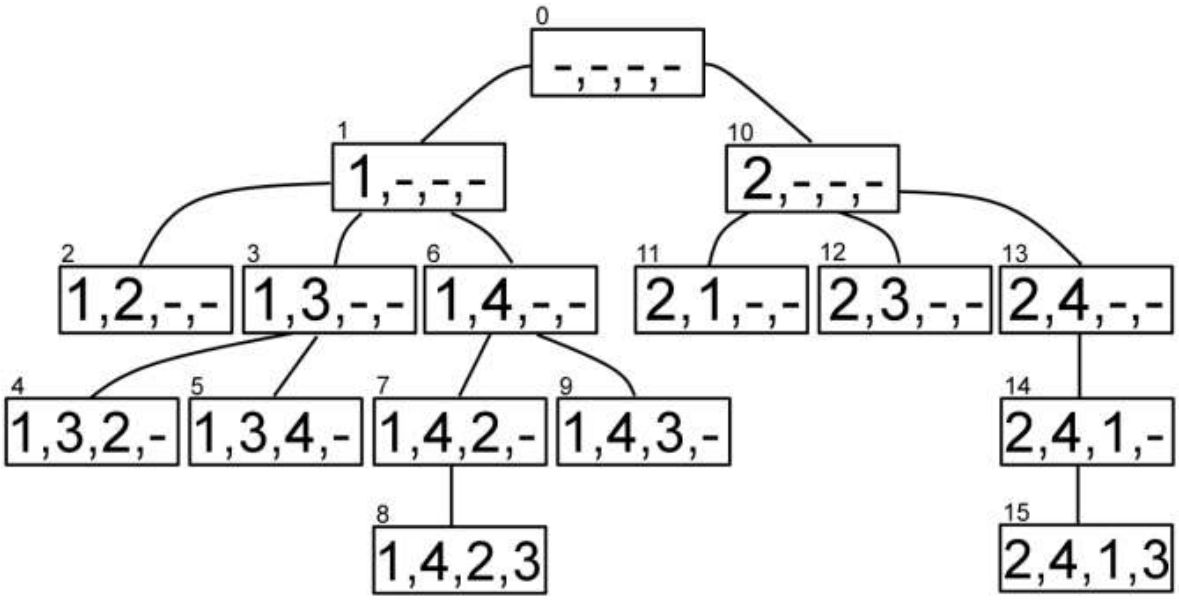
problema, diremos que el tablero es de tamaño 4x4 (ver Figura 4.6) y por lo tanto solo queremos colocar 4 reinas. La solución la iremos almacenando en un arreglo. En cada etapa iremos agregando nodos al árbol de expansión, algunos serán nodos con posibilidad de ser parte de la solución, por lo tanto podremos seguir ramificando el árbol a partir de ellos, y algunos serán nodos fracaso, por lo que habrá que regresar y continuar el camino por otra alternativa. Las restricciones son fácilmente resueltas, la primera de ellas es que no podemos repetir renglones, eso se soluciona si consideramos que cada posición dentro del arreglo solución es un renglón diferente, la segunda restricción es que no podemos repetir columnas, si consideramos que cada índice almacenado en nuestro arreglo representa la columna en la que se colocará cada reina, podemos solucionarlo simplemente no repitiéndolos, la tercer restricción es que no podemos tener reinas en la misma diagonal, esto podemos resolverlo mediante la siguiente ecuación $|x - x^1| \neq |y - y^1|$, en donde (x, y) y (x^1, y^1) son las coordenadas de una reina colocada en el tablero, y la reina que tratamos de colocar.

El tablero sería el siguiente:

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

Figura 4.6 – Tablero de ajedrez 4x4

Con todo esto podemos hallar el siguiente árbol de expansión el cual encuentra una primera solución después haber generado 15 nodos: [5]



La numeración arriba de los nodos deja claro que se trata de una búsqueda primero en profundidad. Siguiendo la lógica del algoritmo de retroceso, se deben explorar todos los nodos en búsqueda de una solución, la cual fue hallada hasta el nodo 15, el nodo 8 aunque completo no es un nodo solución, pues una reina en la posición $(4,3)$ se encontraría atacada por una reina en la posición $(3,2)$.

El nodo solución tiene las reinas en las siguientes posiciones:

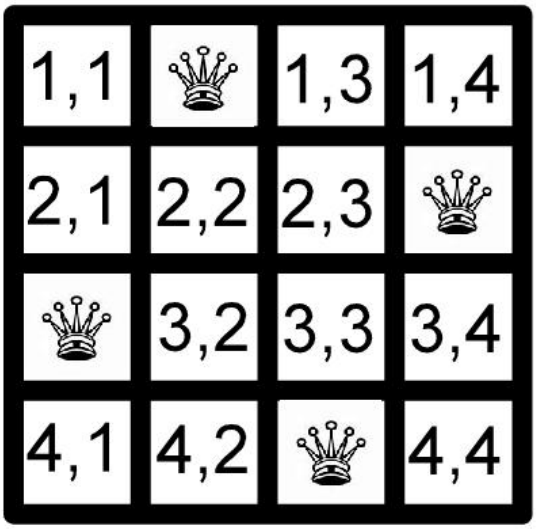


Figura 4.7 – Solución 4 reinas

Como puede verse las reinas no se atacan entre si. Si no se hubiera encontrado una solución habría sido necesario regresar al origen y continuar con el arreglo que comienza en [3,-,-,-].

Con todo esto hemos comprobado que los algoritmos de retroceso, consiguen la solución a base de prueba y error. En el caso de que el retroceso nos llevara hasta la raíz del árbol y no existieran más alternativas que examinar se debe concluir que no se pudo encontrar una solución.

Una vez que tenemos clara la idea del problema mediante el ejemplo anterior (sencillo), podemos buscar la solución al problema normal, un tablero de 8x8 en el cual debemos colocar 8 reinas.

Por la estructura de árbol, lo más apropiado para encontrar las posiciones en donde colocar las 8 reinas, es mediante un algoritmo que emplee recursividad: [4]

Algoritmo: Colocar reina

Entrada: Renglón

Salida: Todas las posibles soluciones

```
for todas las columnas en el renglón do
        if columna esta disponible y las diagonales están disponibles
                Poner la siguiente reina en la columna
                Columna ← No disponible
                Diagonales ← No disponibles
                if renglón < 8
                        ColocarReina(renglón + 1)
                else
                        Éxito, termina la ejecución
                        Remover la reina de la columna, hacer columna y diagonales
                         disponibles
                endif
        endif
endfor
```

Este algoritmo, al igual que otros, pueden encontrarse en los siguientes capitulos programados utilizando el lenguaje C#.

La siguiente imagen muestra una de las posibles soluciones para un tablero de 8x8, la imagen fue obtenida del programa que más adelante deberá ser hecho como un laboratorio, por el lector:

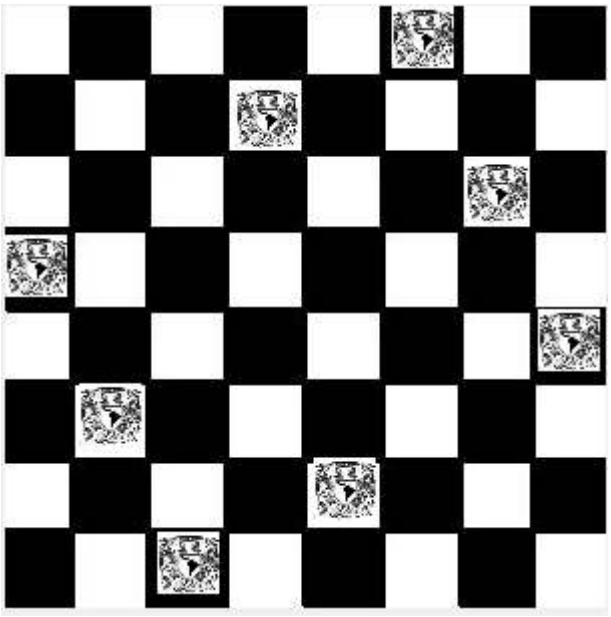


Figura 4.8 – Solución 8 reinas

La complejidad de los algoritmos de retroceso es generalmente de tipo exponencial debido a la búsqueda primero en profundidad, mientras más nodos tengamos mayor la complejidad, es decir generalmente los algoritmos de retroceso $\in O(x^n)$, en donde x serán las posibles opciones que se presenten en cada etapa del algoritmo, y n por lo tanto será la profundidad del árbol. Por lo tanto para el ejemplo de las 8 reinas, sabemos que en cada etapa tenemos por lo menos 8 opciones, y la profundidad del árbol será a lo más de 8. Por lo tanto la complejidad sin restricciones sería $O(8^8)$. Sin embargo al agregar restricciones reducimos considerablemente esta complejidad. Por esta razón, en muchas ocasiones los algoritmos de retroceso se utilizan para hallar la primera solución, y no necesariamente todas, pues esto podría requerir demasiado tiempo.

CAPÍTULO 5

Material de apoyo para el profesor, el alumno y desarrollo de laboratorios

5.1 Material para el profesor

El título de esta tesis menciona el desarrollo de contenido curricular para el tema de algoritmos computacionales.

Para ello decidí generar tres tipos distintos de material, el primero de ellos dirigido a quien dicta la clase: el profesor. Este material es en forma de presentaciones Power Point, de tal manera que pueda sustituirse por las notas de clase y funcionar como guía para el curso.

El material para el profesor lo dividí en 7 módulos, cada uno de los módulos hace uso de una parte de la teoría desarrollada en alguno de los capítulos previos a este.

De esta manera se obtiene la siguiente tabla en la cual queda claro de donde se tomó la teoría empleada para desarrollar los módulos del material curricular:

MÓDULO	CAPÍTULO TESIS
MÓDULO 0	CAPÍTULO 2
MÓDULO 1	CAPÍTULO 3
MÓDULO 2	CAPÍTULO 3
MÓDULO 3	CAPÍTULO 4
MÓDULO 4	CAPÍTULO 4
MÓDULO 5	CAPÍTULO 4
MÓDULO 6	CAPÍTULO 4

La razón de separar algunos capítulos en varios módulos fue básicamente por practicidad, para poder crear la idea de que se termina un tema y dar al alumno la sensación de que concluye un ciclo, y además de esta manera poder aplicar una prueba mediante la cual medir de alguna manera que tanto se ha aprendido.

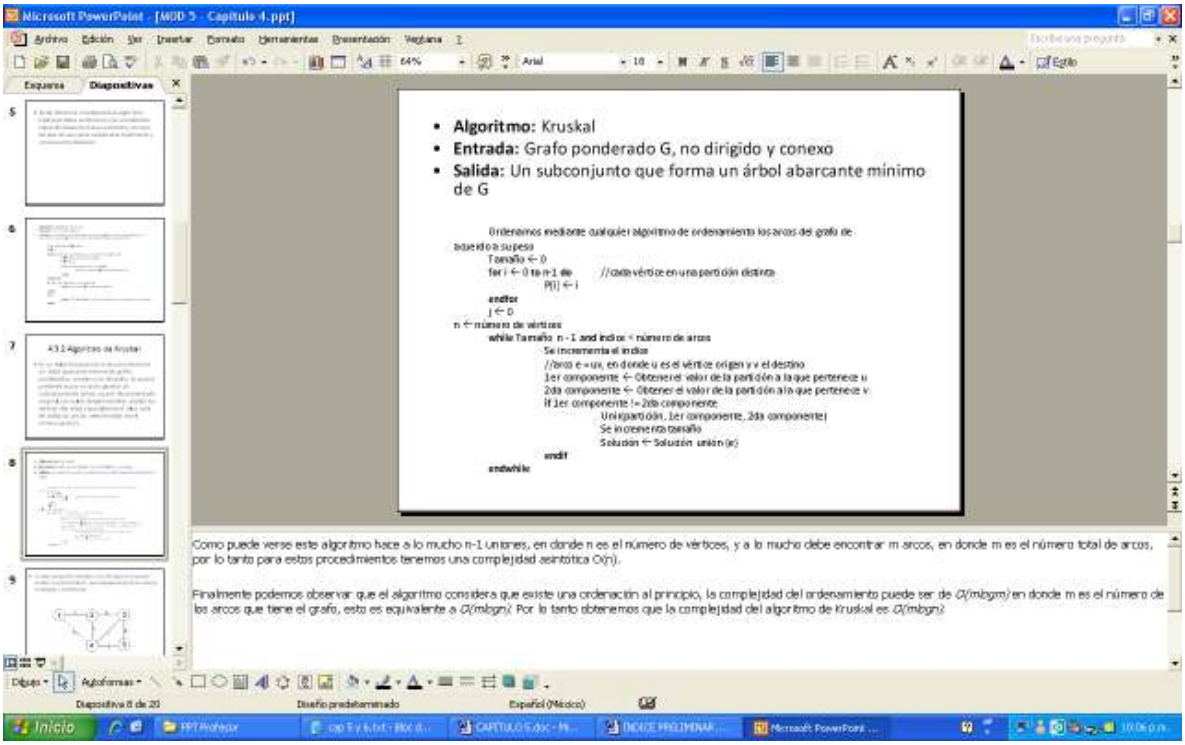
Además de la teoría en las presentaciones, se incluyó también en forma de Notas dentro de las mismas presentaciones Power Point, información que el profesor puede consultar para complementar su conocimiento y así exponer clases más ricas en contenido.

El material del profesor fue desarrollado todo bajo un mismo esquema, de forma que los 7 módulos en si forman un solo curso que puede seguirse de forma regular.

Haciendo analogía a un curso impartido en nuestra facultad; el material presentado en esta tesis podría ser utilizado en una clase de 9 créditos con duración total de 72 horas, el tiempo sugerido para cada tema quedaría como sigue:

TEMA	HORAS
Módulo 0 – Microsoft Visual Studio 2005 y C#	6
Módulo 1 – Algoritmos y Complejidad	10
Módulo 2 – Análisis asintótico y ejemplos	14
Módulo 3 – "Divide y vencerás"	14
Módulo 4 – Programación dinámica	10
Módulo 5 – Algoritmos codiciosos	10
Módulo 6 – Algoritmos de retroceso	8
	Total: 72 horas

Las presentaciones a disposición del profesor tienen el formato que se muestra en la siguiente figura:



Como puede verse, tienen tanto una parte que será expuesta al alumno como las mencionadas notas que se encuentran en la parte inferior.

El material para profesor cuenta con un **total de 231 diapositivas** distribuidas de la siguiente manera:

MÓDULO	NÚMERO DE DIAPOSITIVIAS
Módulo 0	61
Módulo 1	25
Módulo 2	49
Módulo 3	44
Módulo 4	18
Módulo 5	20
Módulo 6	14

Las presentaciones se encuentran en el CD-ROM adjunto, dentro de la carpeta:

\ PPT Profesor

5.2 Material para el alumno

El material destinado para el alumno fue desarrollado con las siguientes características en mente:

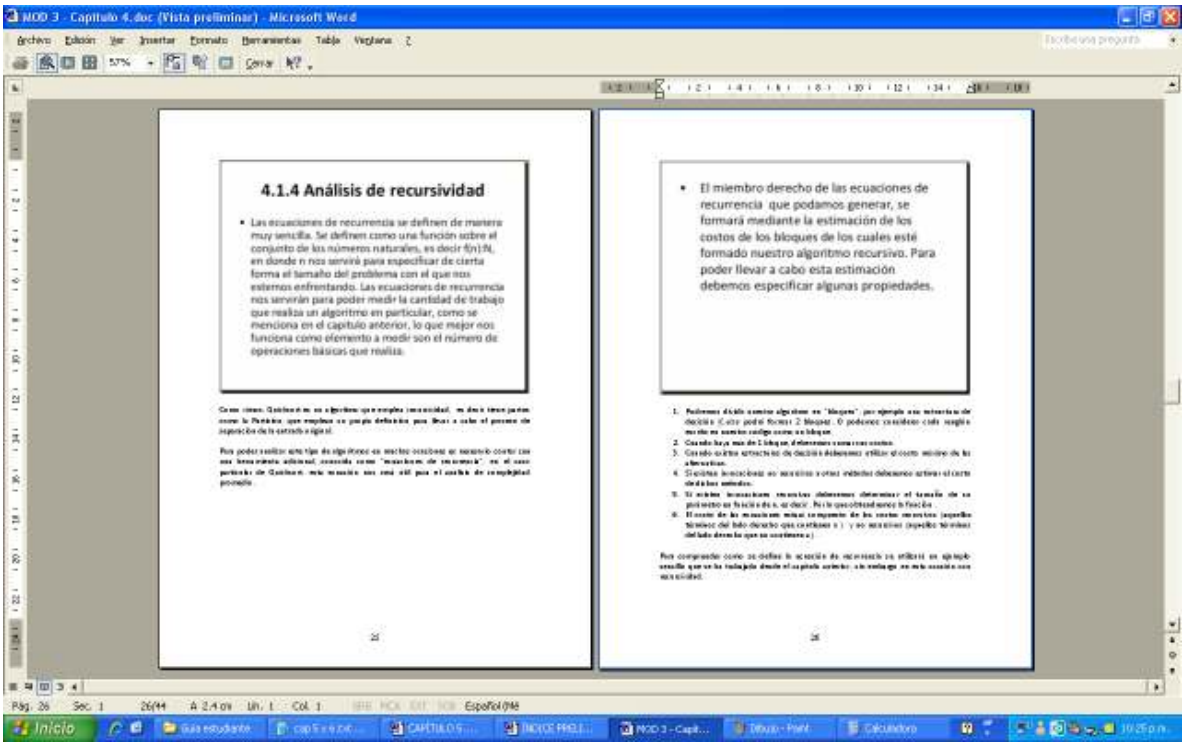
- Funcionar como apuntes de referencia.
- Ser una guía que le permita al alumno seguir la exposición del profesor de forma más sencilla.
- Obtener información extra referente al tema que se esté exponiendo.
- El material para el alumno es de tipo autocontenido por lo tanto puede utilizarse como medio de estudio.

Al igual que el material para el profesor, el material para el alumno se encuentra dividido en los mismos 7 módulos bajo la misma lógica expuesta anteriormente.

El alumno dispone de una imagen reducida de la diapositiva que el profesor expone, y en la parte inferior información complementaria que servirá como apuntes y podrán perfectamente complementar la información de clase.

El formato utilizado para estas guías es .doc de Word.

Las guías para el alumno tienen el siguiente formato.



Como se puede apreciar, en la parte superior se encuentra una imagen tomada directamente de las diapositivas del material del profesor y en la parte inferior se encuentra información suplementaria para consulta del alumno. Se pretende que el alumno pueda utilizar este material para estudiar, apoyando así que se concentre en la clase y no en escribir o que pueda únicamente hacer las anotaciones que considere más importantes.

El material para el alumno cuenta con un **total de 233 páginas** distribuidas de la siguiente manera:

MÓDULO	NÚMERO DE PÁGINAS
Módulo 0	63
Módulo 1	25
Módulo 2	49
Módulo 3	44
Módulo 4	18
Módulo 5	20
Módulo 6	14

Las guías para el alumno se encuentran en el CD-ROM adjunto, dentro de la carpeta:

\ Guía estudiante

5.2 Desarrollo de los laboratorios

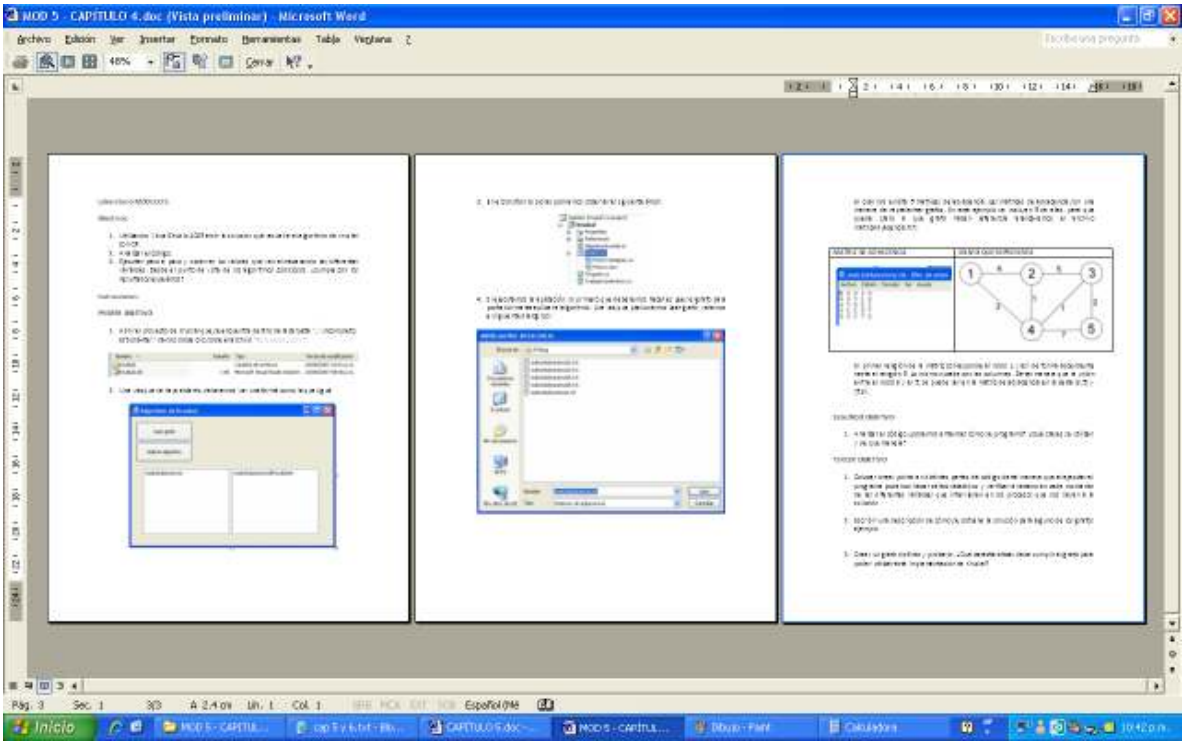
Al final de cada módulo, el alumno deberá completar además de una serie de preguntas, un laboratorio.

Un laboratorio es una aplicación de la teoría aprendida, en la cual deberá llevar a cabo una serie de tareas, en todas empleando la herramienta de trabajo Visual Studio 2005, y el lenguaje de programación C#.

Existen en total 7 laboratorios, distribuidos entre los módulos.

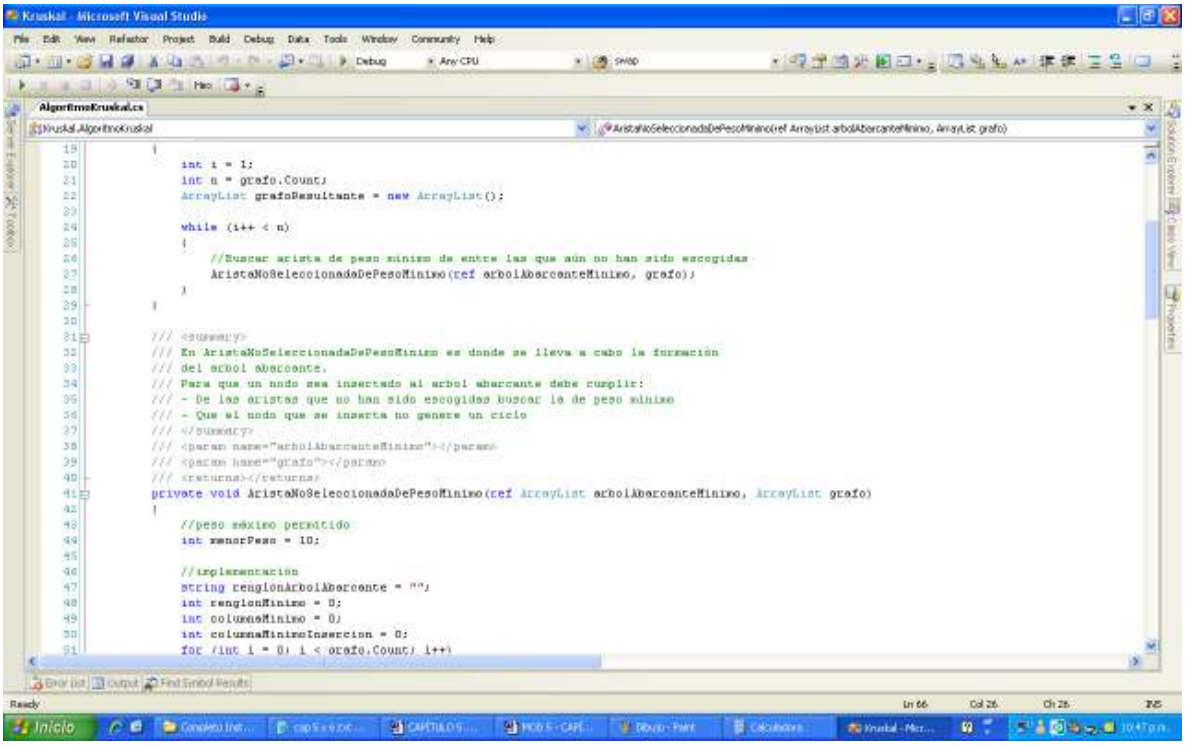
Los laboratorios cuentan con una guía de desarrollo. Las guías presentan una serie de objetivos y a su vez una serie de pasos que se deberán seguir para completarlos. Los objetivos son muy distintos entre si, en el primer módulo por ejemplo hay que crear una solución desde cero a un problema dado, mientras que en algunos otros habrá que completar solo ciertas partes del código, o agregar cierta funcionalidad.

Las guías para el estudiante se ven de la siguiente manera:



Así mismo, se distribuye para el profesor la solución a los 7 laboratorios, es decir el archivo de solución para Visual Studio el cual incluye además de las formas todo el código fuente necesario para la ejecución del programa.

El código fuente con la solución se encuentra perfectamente comentado, de forma que al profesor se le hará muy fácil entenderlo:



Los laboratorios se encuentran en el CD-ROM adjunto, dentro de la carpeta:

\ Laboratorios por hacer

Y a su vez dentro de la carpeta que corresponda a cada módulo. Dentro de las carpetas de cada módulo encontraremos un archivo Word el cual contiene la guía a seguir, además de otras 2 carpetas, una con la solución completa destinada al instructor y otra con la solución parcial destinada al estudiante. Solamente en el módulo 0 no se incluye solución parcial para el estudiante.

CONCLUSIONES

El principal objetivo de esta tesis fue redactar sobre un tema que parece difícil, quizá incluso oscuro para muchos de nosotros de una forma tal que pudiera ser comprendido fácilmente y que nos iniciara entre los que programan de forma eficiente; considero que lo logré, al releer y estudiar lo que escribí me doy cuenta que conseguí aplicar toda la teoría de forma concisa, y que si alguien consulta este material encontrará en él un texto sencillo y sin embargo muy completo en temas.

Los ejercicios desarrollados son ejemplos claros de cada tema, que llevan al lector de la mano, y que reafirman la teoría dejando en el que lee una clara idea de lo que se trata analizar un algoritmo.

Al final de esta tesis, puedo concluir con seguridad que el análisis de algoritmos no se aplica únicamente a productos de software con orientación científica, bioinspirada o de índoles meramente académicas, el análisis de algoritmos puede servir para mejorar muchos de los procesos cotidianos en una aplicación de producción que se pudiera utilizar en cualquier empresa, pudiendo llevar a que los procesos se lleven a cabo de una forma mucho más eficiente y quizá incluso mejorar el rendimiento de entidades secundarias como pudiera ser una red de datos o un servidor al cual muchos clientes se conectan.

En el caso más extremo, el análisis de algoritmos pudiera aplicarse a la vida cotidiana, pues si nos vamos a la definición más simple de algoritmo, cada tarea que realizamos y para la cual necesitamos una serie de pasos podría verse analizada por las herramientas matemáticas aquí descritas.

El análisis de algoritmos debe ser uno de los pilares centrales en la formación de un Ingeniero en Computación que finalmente acabará en la mayoría de las veces llevando sus ideas a unas cuantas líneas de código, y que mejor si puede validarlas e incluso mejorarlas.

Existen muchos problemas y como iniciación vale la pena revisar a los autores que han trabajado en muchos de ellos, de tal manera que al terminar de estudiar las principales estrategias para el diseño de algoritmos queda claro que a cada problema hay que estudiarlo y atacarlo bajo la mejor estrategia o de lo contrario acabaremos con un resultado pobre, o en el peor de los casos acabaremos sin obtener un resultado.

Definitivamente al terminar este trabajo, he generado un contenido curricular en el tema de Análisis y Diseño de Algoritmos Computacionales de mucha utilidad para el que imparta la materia en nuestra facultad o en cualquier otra. Es un texto que también puede considerarse autocontenido pues no necesariamente debe revisarse bajo el marco de un curso o la supervisión de un profesor, pues toda la información principal esta escrita y detallada en el mismo trabajo.

El curso (los módulos para profesor, alumno y laboratorios) se pondrán a disposición del mundo de forma gratuita a través de un sitio Web, sin embargo la mejor de las opciones

sería generar un sitio en el cual pueda ser consultada la información y visualizar los ejemplos únicamente con un navegador Web. Expandir este proyecto para cumplir con ese objetivo es ya un hecho (se desarrollará durante los próximos meses y en base a este trabajo), de esta manera se pretende que universidades con pocos recursos puedan capacitarse sin necesidad de invertir.

Además de esto es importante mencionar, que este proyecto de tesis se pretende modificar de manera que se adecue como apuntes de la facultad.

Finalmente, en base a la experiencia obtenida durante el desarrollo de esta tesis y al material curricular generado, yo recomendaría que el curso que actualmente se imparte en la facultad y que abarca el tema de algoritmos y estructuras de datos se separe en dos, el análisis de algoritmos y el estudio de las principales estrategias de diseño de algoritmos son tan vitales en la formación de un Ingeniero en Computación que merece un curso completo; en el capítulo 5 de esta tesis se propone una distribución de horario en base a los módulos del material curricular generado.

GLOSARIO

.NET

.NET es un proyecto de Microsoft para crear una nueva plataforma de desarrollo de software con independencia de plataforma y que permita un rápido desarrollo de aplicaciones. Basado en esta plataforma, Microsoft intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el Sistema Operativo hasta las herramientas de mercado.

.NET Framework

Es un entorno para desarrollar y ejecutar aplicaciones Web, Servicios Web, y otro tipo de aplicaciones. Consiste de 3 partes principales: El CLR, la librería de clases base, ASP.NET, ADO.NET, XML y Windows Forms.

ADO.NET

Es lo más reciente en una extensa línea de tecnologías de acceso a bases de datos que comenzó hace varios años con la interfaz de programación de aplicaciones de la conectividad abierta de base de datos ODBC. ADO.NET nos provee de una serie de objetos que nos facilitan el acceso a datos en nuestras aplicaciones .NET.

AMD

Fundada en 1969 y con su central situada en Sunnyvale, California, USA, Advanced Micro Devices (AMD) es la segunda compañía mundial productora de microprocesadores (detrás de Intel) y uno de los más importantes fabricantes de memoria flash y otros dispositivos semiconductores.

AMD64

Representa la entrada de AMD dentro del mercado de los microprocesadores de 64 bits.

Árbol

Estructura de datos ampliamente usada que emula la forma de un árbol (un conjunto de nodos conectados). Un nodo es la unidad sobre la que se construye el árbol y puede tener cero o más nodos hijos conectados a él. Se dice que un nodo A es padre de un nodo B si existe un enlace desde A hasta B (en ese caso, también decimos que B es hijo de A). Sólo puede haber un único nodo sin padres, que llamaremos raíz. Un nodo que no tiene hijos se conoce como hoja.

Árbol abarcante mínimo

Es una estructura que contiene el mínimo número de arcos que conectan a todos los vértices de un grafo conexo. Por lo tanto debe contener a todos los vértices del grafo original.

ASP.NET

Se puede utilizar para desarrollar sitios Web dinámicos, aplicaciones Web en general, servicios Web XML. Forma parte de la plataforma .NET de Microsoft. ASP.NET nos provee de una serie de objetos que nos permiten interactuar con un cliente y generar de esta manera contenido Web dinámico.

CLR

Common Language Runtime. Es el componente que hace las veces de máquina virtual para la plataforma .NET de Microsoft. Es una tecnología que define el entorno sobre el cual se debe ejecutar el código de un programa. El CLR provee de todos los requisitos que puede llegar a necesitar una aplicación .NET.

COM

Component Object Model. Es una tecnología desarrollada por Microsoft, su objetivo principal es crear componentes de software. Cada componente COM puede ser entonces utilizado en entornos distintos al cual fueron creados.

COM+

Es una extensión de COM. Provee de soporte para transacciones distribuidas, pooling de recursos, aplicaciones desconectadas, mejor manejo de memoria y procesadores entre otras cosas.

Console Application

En Visual Studio podemos crear distintos tipos de aplicaciones basadas en Windows, una de ellas es Console Application la cual utiliza un entorno tipo consola para su ejecución.

Diagrama de Venn

Son diagramas utilizados en la teoría matemática de conjuntos. En ellos se muestran todas las posibles relaciones matemáticas y/o lógicas entre dos conjuntos.

Dígrafo

Dígrafo o grafo dirigido. Cumplen con la misma definición de un grafo salvo que deben tener una dirección establecida.

ECMA

European Computer Manufacturers Association. Es una organización internacional para estándares en sistemas de información y comunicación.

EM64T

Representa la implementación de Intel para microprocesadores de 64 bits.

FIFO

First In, First Out. Primero en entrar, Primero en salir. Esta expresión representa a las estructuras que sirven datos con el siguiente principio: El primer elemento que se haya recibido será el primer elemento que sea despachado, lo que venga después tendrá que esperar a que se termine con el primer elemento.

Grafo

Es un conjunto de puntos, nodos o vértices conectados mediante líneas o arcos. Un grafo no es dirigido por omisión.

Grafo conexo

Un grafo es conexo si y solo si existe un camino simple en cualesquiera dos nodos del grafo.

IA-64

Representa al conjunto de servidores de Intel denominados Itanium. Microprocesadores de 64 bits para uso principalmente empresarial y para computadoras de alto rendimiento.

IDE

Integrated Development Environment. Es un software computacional que asiste a los programadores en el desarrollo de software.

Intel

Principal compañía productora de semiconductores. Fundada en 1968 como Integrated Electronics Corporation, con base en Santa Clara, California, USA.

ISO

International Organization for Standarization . Es un cuerpo internacional para el establecimiento de estándares. Fundado en 1947, la organización se dedica a producir estándares mundiales comerciales e industriales.

Java

Lenguaje para desarrollar aplicaciones de diversos tipos, enfocado para programación orientada a objetos. Es un lenguaje de alto nivel desarrollado por Sun Microsystems a principios de los 90s.

LIFO

Last In, First Out. Último en entrar, Primero en salir. Esta expresión representa a las estructuras que sirven datos con el siguiente principio: El último elemento que ingrese a la estructura, será el primero que sea despachado, el penúltimo elemento se despachará después que se termine con el primero.

Microsoft

Multinacional dedicada principalmente a la producción de software. En la actualidad también tienen productos de hardware. Es la encargada de desarrollar la plataforma .NET así como el entorno de desarrollo Visual Studio. Tiene su ubicación en Redmond, Washington, USA. Su producto más vendido es el sistema operativo Windows.

MSDN

Microsoft Developers Network. Es la parte responsable de manejar las relaciones con los desarrolladores. Incluye suscripciones para descargas de contenido que pueden ir desde software hasta notas técnicas.

Recursividad

Es una manera de definir métodos o funciones en las cuales el método o función se especifica basándose en términos de su propia definición.

SDK

Software Development Kit. Comúnmente es un conjunto de herramientas que permiten al desarrollador de software crear aplicaciones para un cierto paquete de software, plataforma de hardware, sistema computacional, consola de video juegos, sistema operativo, etc.

SQL

Structured Query Language. Lenguaje computacional usado para crear, obtener, actualizar y borrar datos de una base de tipo relacional como por ejemplo SQL Server 2005.

Visual Basic, C#, J#, C++

Lenguajes de programación de alto nivel orientado objetos y a eventos. Forma parte de los lenguajes “básicos” de la plataforma .NET, se incluyen junto con Visual Studio 2005 Professional. Por ser lenguajes de alto nivel permiten un desarrollo de aplicaciones muy rápido.

Windows Application, Windows Forms

En Visual Studio podemos crear distintos tipos de aplicaciones basadas en Windows, una de ellas

es Windows Application la cual utiliza el entorno gráfico de Windows, de tal manera que podemos visualizar controles típicos de una ventana Windows como pueden ser botones, cajas de texto, barras de desplazamiento, etc. Es ideal para crear aplicaciones ricas en contenido.

XML

Extensible Markup Language. Es un lenguaje de propósito general. Su principal objetivo hacer más fácil el compartir información a través de diferentes sistemas de información, particularmente vía Internet.

BIBLIOGRAFÍA Y MESOGRAFÍA

[1] Sedgewick, Robert
"Algorithms"
Editorial Addison-Wesley
1983

[2] Wirth, Niklaus
"Algorithms and Data Structures"
Editorial Prentice-Hall
1985

[3] Berman, Kenneth A. y Paul, Jerome L.
"Algorithms: Sequential, Parallel, and Distributed"
Edtorial Thomson
2005

[4] Drozdek, Adam
"Data structures and algorithms in C++"
Editorial Thomson
2005

[5] Baase, Sara y Van Gelder Allen Van
"Algoritmos computacionales, Introducción al análisis y diseño"
Editorial Pearson Educación
2002

[6] Parker, Alan
"Algorithms and Data Structures in C++"
Editorial CRC Press
1993

[7] Baldwin, Douglas
"Algorithms and Data Structures: The Science of Computing"
Editorial Thomson
2004

[8] Sedgewick, Robert
"Algorithms in Java"
Editorial Addison-Wesley
2002

[9] Dale, Nell
"C++ Data Structures"
Editorial Jones and Bartlett Publishers
2003

[10] Weiss, Mark
"Data Structures and Problem Solving Using C++"
Editorial Addison-Wesley
2002

[11] Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford
"Introduction to Algorithms"
Editorial MIT Press
2001

[12] Ferguson, Jeff
"C# Bible"
Editorial Wiley Publishing
2002

[13] Champlain, Michel
"C# 2.0 Practical Guide for Programmers"
Editorial Morgan Kaufmann Publishers
2005

[14] Sharp, John
"Microsoft Visual C# 2005, Step by Step"
Editorial Microsoft Press
2006

[15] Microsoft; "Productos – Visual Studio"
http://msdn2.microsoft.com/en-us/vstudio/aa700919.aspx

[16] Microsoft; "What's New in the .NET Framework Version 2.0"
http://msdn2.microsoft.com/en-us/library/t357fb32.aspx

[17] Microsoft; ".NET Compact Framework"
http://msdn2.microsoft.com/en-us/library/f44bbwa1.aspx

[18] Microsoft, "ADO.NET para el programador de ADO"
http://www.microsoft.com/spanish/msdn/articulos/archivo/180501/voices/adonetdev.asp