Universidad Mayor de San Andrés
Facultad de Ciencias Puras y Naturales
Carrera de Informática

Fundamentos de la Programación

Jorge Humberto Terán Pomier
<teranj@acm.org>

2º Edición - Abril 2010
1º Edición - Diciembre 2006
Nro. Depósito Legal 4-1-116-10PQ
ISBN:978-99954-0-820-6
La Paz - Bolivia

Prefacio

Motivación

Los conceptos presentados se estudian en muchas de las materias del pregrado de la carrera de informática, con el rigor necesario. Sin embargo, en mi experiencia docente he podido ver que hay muchos estudiantes que carecen de elementos prácticos con los que puedan aplicar directamente los conceptos aprendidos y lo que se pretende es cubrir este vacío.

El presente texto trata de introducir los conceptos de programación, a estudiantes de pregrado con un enfoque nuevo.

Para una adecuada compresión de la temática presentada en el libro, el lector debrá tener alguna experiencia en el desarrollo de programas. El libro no pretende enseñar a codificar programas, sino a resolver problemas.

Contenido

Se presentan en el capítulo 1, los conceptos de algorítmica elemental con el propósito de introducir a los estudiantes, al concepto de la eficiencia de los programas.

El capítulo 2 introduce los conceptos de análisis de algoritmos con una serie de ejercicios aplicados. Se incluyen algunos laboratorios que tienen el propósito de que, los estudiantes asimilen estos conceptos en forma experimental.

El capítulo 3 introduce el concepto de la teoría de números, introduciendo las librerías de Java. Se muestran ejemplos de números grandes y aplicaciones en la criptografía.

El capítulo 4 trata de la escritura de programas con miras a la prueba del código. En este capítulo se introduce un concepto que es el diseño por con-

tratos y la forma de aplicar en Java. Aunque este concepto fue introducido inicialmente en el lenguaje Eifel, actualmente tiene mucho interés en el desarrollo de aplicaciones. El texto no trata de trabajar en conceptos de lógica formal en su totalidad, lo que propone es una introducción a estos conceptos para facilitar la prueba del código.

El capítulo 5 complementa el capítulo 4 con conceptos de clasificación y búsqueda . Se introducen laboratorios y las bibliotecas Java para clasificar.

El capítulo 6 está orientado a explicar como se encara la programación de problemas de combinatoria básica.

El capítulo 7 explica como utilizar las librerías del lenguaje Java y como resolver problemas con pilas, listas enlazadas, conjuntos, árboles y colas de prioridad.

EL capítulo 8 introduce al estudiante a los problemas de retroceso (bactracking) con problemas típicos, como recorrido de grafos y permutaciones.

El capítulo 9 introduce a la geometría computacional. Muestra como construir una librería de rutinas básicas para desarrollar aplicaciones relacionadas a la geometría y muestra las librerías Java así como las posibilidades que se tienen con las mismas.

El lenguaje que se escogió fue el Java, primero porque la formación de la Universidad Mayor de San Andrés está orientada a este lenguaje. Segundo porque el lenguaje es de amplia utilización en las empresas y en la educación.

Aplicación en el aula

Esta temática puede aplicarse en el transcurso de un semestre en clases teóricas, prácticas y de laboratorio.

Los laboratorios proponen ejercicios que tienen la finalidad de evaluar experimentalmente la ejecución de los algoritmos. Esto permite que los estudiantes obtengan una vivencia de los resultados teóricos.

Se puede aplicar en una clase de teoría de dos períodos y una segunda clase que sería de ejercicios o laboratorio.

Agradecimientos

Quiero agradecer a mi esposa, a mis hijos por los aportes realizados finalmente al Dr. Miguel Angel Revilla por dejarme utilizar ejercicios del Juez en

Línea en los ejemplos del libro.

Notas de la segunda edición

La segunda edición corrige varios errores que se detectaron en varios capítulos del libro. Se han mejorado los ejercicios propuestos presentando una lista más amplia para cada tema.

A solicitud de muchos estudiantes se ha agregado un apéndice que explica como obtener una cuenta en el Juez en línea y la forma de resolver un ejercicio en Java. Se explica la forma de realizar la entrada y salida de datos y algunas técnicas para mejorar la eficiencia de un programa.

Los enunciados que figuran al final de capítulo del libro se cambiaron por una lista de problemas que pueden resolverse con los métodos planteados. Estos problemas se encuentran en el sitio de Valladolid.

Un aspecto importante es la utilización del sitio http://uvatoolkit.com/ que provee ayudas muy útiles para resolver los problemas planteados. Los problemas al final de cada capítulo siguen la clasificación provista en este sitio.

Programas presentados en el libro

El lector puede acceder a todos los programas presentados en el texto en el sitio web http://www.icpc-bolivia.edu.bo. Hay que aclarar que todos han sido codificados utilizando el compilador Java 1.6 de Sun Microsystems. Los programas fueron realizados utilizando el editor Eclipse por lo cual debería ser simple acceder a los programas.

M.Sc. Jorge Terán Pomier.

Índice general

Prefacio III

1. Algorítmica elemental 1
- 1.1. Introducción . . . 1
- 1.2. Algoritmo . . . 1
- 1.3. Problemas e instancias . . . 2
- 1.4. Eficiencia . . . 2
- 1.5. Qué considerar y qué contar . . . 5
  - 1.5.1. Operaciones elementales . . . 6
  - 1.5.2. Análisis del mejor caso, caso medio y peor caso . . . 6
  - 1.5.3. Ejercicios . . . 7
  - 1.5.4. Laboratorio . . . 7

2. Análisis de algoritmos 11
- 2.1. Introducción . . . 11
- 2.2. Notación orden de . . . 11
  - 2.2.1. Propiedades de O grande de n . . . 12
- 2.3. Notación asintótica condicional . . . 13
- 2.4. Notación omega . . . 13
- 2.5. Notación teta . . . 13
- 2.6. Análisis de las estructuras de control . . . 14
  - 2.6.1. Secuenciales . . . 14
  - 2.6.2. Ciclos For . . . 14
  - 2.6.3. Llamadas recursivas . . . 14
  - 2.6.4. Ciclos while y repeat . . . 15
  - 2.6.5. Análisis del caso medio . . . 16
  - 2.6.6. Análisis amortizado . . . 17
- 2.7. Solución de recurrencias . . . 18

2.7.1. Método por substitución 18
2.7.2. Cambio de variables 19
2.7.3. Ejercicios 20
2.7.4. Método iterativo 20
2.7.5. Ejercicios 22
2.7.6. Teorema master 22
2.7.7. Solución general de recurrencias lineales 23
2.8. Ejercicios resueltos 27
2.9. Ejercicios 32
2.10. Nota sobre el cálculo integral 34

3. Teoría de números **35**
3.1. Introducción 35
3.2. Variables del lenguaje Java 35
3.3. Cálculo del máximo común divisor 38
3.3.1. Divisibilidad 39
3.3.2. Algoritmos extendido de Euclides 42
3.4. Mínimo común múltiplo 44
3.5. Aritmética modular 44
3.5.1. Propiedades 45
3.5.2. Congruencias 46
3.5.3. Inverso multiplicativo 46
3.5.4. Solución de ecuaciones 46
3.6. Números primos 48
3.6.1. Generación de primos 48
3.6.2. Factorización 50
3.6.3. Prueba de la primalidad 52
3.6.4. Teorema de Fermat 53
3.6.5. Prueba de Miller - Rabin 53
3.6.6. Otras pruebas de primalidad 54
3.7. Bibliotecas de Java 54
3.7.1. Ejemplo 55
3.8. Ejemplos de aplicación 58
3.8.1. Ejercicios 65

4. Codificación con miras a la prueba **67**
4.1. Introducción 67
4.2. Algunos errores de programación 68

4.3. Especificación de programas 72
4.3.1. ¿Por qué especificar? 74
4.3.2. ¿Qué especificar? 75
4.3.3. ¿Cómo especificar? 76
4.3.4. Invariantes 77
4.3.5. Ejercicios propuestos 80
4.4. Aplicaciones de las invariantes 81
4.4.1. Principio de correctitud 90
4.5. Diseño por contratos 91
4.5.1. Especificación de contratos 94
4.5.2. Invariantes 96
4.6. Prueba estática 97
4.7. Prueba dinámica 98
4.7.1. Afirmaciones 98

5. Aplicaciones de búsqueda y clasificación 103
5.1. Introducción 103
5.2. Algoritmos de búsqueda 103
5.2.1. Prueba exhaustiva 107
5.2.2. Representación gráfica de la búsqueda 109
5.3. Clasificación 114
5.3.1. Clasificación en Java 115
5.3.2. Algoritmos de clasificación 119
5.3.3. Laboratorio 133
5.3.4. Ejemplo de aplicación 134
5.3.5. Ejemplo 1 134
5.3.6. Ejemplo 2 137
5.4. Ejercicios 140

6. Combinatoria básica 141
6.1. Introducción 141
6.2. Técnicas básicas para contar 141
6.3. Coeficientes binomiales 143
6.4. Algunas secuencias conocidas 145
6.4.1. Números de Fibonacci 145
6.4.2. Números Catalanes 146
6.4.3. Número Eulerianos 147
6.4.4. Números de Stirling 150

6.4.5. Particiones enteras 151
6.4.6. Ejemplo de aplicación 153
6.5. Ejercicios 156

7. Estructuras de datos elementales **157**
7.1. Introducción 157
7.2. Vectores 157
7.2.1. Vectores estáticos 157
7.2.2. Vectores dinámicos 158
7.3. Pilas 162
7.4. Listas enlazadas 166
7.5. Conjuntos 169
7.6. Clases map 171
7.7. Clases para árboles 173
7.8. Clases para colas de prioridad 175
7.9. Ejercicios para laboratorio 178
7.10. Ejercicios para el Juez de Valladolid 178

8. Backtracking **179**
8.1. Introducción 179
8.2. Recorrido de grafos 179
8.3. Constuir todos los subconjuntos 183
8.4. Construir todas las permutaciones 187
8.5. Ejemplo de Aplicación 191
8.6. Ejercicios 194

9. Geometría computacional **195**
9.1. Introducción 195
9.2. Geometría 196
9.3. Librerías de Java 202
9.4. Cercos convexos 204
9.5. Clase Java para polígonos 211
9.6. Cálculo del perímetro y área del polígono. 212
9.7. Ejercicios 216

A. Fundamentos matemáticos **219**
A.1. Recordatorio de logaritmos 219
A.2. Recordatorio de series 220

A.2.1. Series simples 220
A.2.2. Serie aritmética 220
A.2.3. Serie geométrica 220
A.2.4. Propiedades de la sumatorias 221
A.2.5. Series importantes 221
A.3. Combinatoria básica 222
A.3.1. Fórmulas importantes 222
A.4. Probabilidad elemental 223
A.5. Técnicas de demostración 224
A.5.1. Demostración por contradicción 224
A.5.2. Demostración por inducción 224

B. El Juez de Valladolid **227**
B.1. Como obtener una cuenta 227
B.2. Como enviar un problema 231
B.3. Problema de ejemplo 233
B.4. Ayuda para corregir errores 236
B.5. Ejercicios 237

Capítulo 1

Algorítmica elemental

1.1. Introducción

En este capítulo empezamos el estudio de los algoritmos. Empezaremos definiendo algunos términos tales como algoritmo, problema, instancia, eficiencia e investigaremos métodos para probar la eficiencia de los mismos.

Existen métodos formales para realizar pruebas rigurosas sobre la corrección de los programas ésta temática está fuera del alcance del texto pero será mencionada más adelante solo como referencia.

1.2. Algoritmo

Definimos como algoritmo a un conjunto de pasos necesarios para resolver un problema ya sea manualmente o por métodos mecanizados que, son los más usuales en la actualidad. El concepto de algoritmos fue definido inicialmente por el matemático Persa Al-Khowârizmî en el siglo diecinueve.

La ejecución de un algoritmo no debe incluir procedimientos intuitivos o que requieran creatividad. Por ejemplo, una receta de cocina puede denominarse un algoritmo si contiene las instrucciones precisas para preparar algo, siempre y cuando no tenga detalles tales como sal al gusto, o cocinar hasta que esté suave, que son apreciaciones subjetivas del que prepara la receta.

Debemos indicar que los algoritmos deben cumplir un requisito fundamental y es que todo algoritmo debe terminar en un número finito de pasos. Si consideramos el sistema operativo veremos que no es un algoritmo porque no termina nunca, dado que está en un ciclo infinito.

Podemos citar algunos ejemplos de algoritmos conocidos, el método de multiplicar que aprendimos en la escuela, el algoritmo para verificar si un número es primo y muchos otros.

1.3. Problemas e instancias

Si pensamos los procedimientos para multiplicar números que conocemos veremos que hay varias formas de resolver el problema de multiplicación. Tenemos los métodos que aprendimos en la escuela, métodos por división y multiplicación por 2 denominado multiplicación a la rusa y otros. ¿Cuál de éstas posibles soluciones hay que implementar? Esta decisión corresponde a un área muy desarrollada en el campo de la informática denominada análisis de algoritmos.

Una instancia particular sería por ejemplo multiplicar el número 123 por el 4567 pero un algoritmo debe ser capaz de funcionar correctamente no solo en una instancia del método sino más bien en todas las instancias posibles.

Para demostrar que un algoritmo es incorrecto, es suficiente demostrar que es incorrecto para una instancia . Así como es difícil probar que un teorema es correcto también es difícil probar que un algoritmo es correcto.

Existen limitaciones propias de las computadoras que ponen restricciones a los algoritmos. Estas pueden ser debido al tamaño de los números, espacio de memoria o almacenamiento. En el análisis de los algoritmos, se trata de analizarlos en forma abstracta inicialmente, pero en el transcurso del texto también tocaremos aspectos específicos sobre la implementación y la eficiencia de los mismos.

1.4. Eficiencia

Cuando tenemos que resolver un problema pueden existir muchos algoritmos disponibles. Obviamente quisiéramos escoger el mejor. De ahí nos viene la pregunta ¿Cuál es mejor?

Si tenemos un problema sencillo con algunas pocas instancias escogemos, lo más fácil y nos despreocupamos de la eficiencia.

Para analizar ésto tenemos dos métodos. El método *empírico* también llamado a posteriori, consiste en programar todos los métodos y probarlos con diferentes instancias y con la ayuda de la computadora analizamos y

escogemos alguno.

El segundo llamado método apriori es el análisis *teórico* del algoritmo, que con la ayuda de las matemáticas podemos determinar la cantidad de los recursos requeridos por cada algoritmo en base del tamaño de las instancias consideradas.

El tamaño de una instancia normalmente está definida por el número de bits, en el caso de un grafo por el número de nodos y vértices, o en otros por el número de elementos a procesar. La ventaja del método teórico es que no depende de la computadora, lenguaje de programación o implementación particular.

En el tratamiento del tema utilizaremos en algunos casos un método híbrido donde se determinará la eficiencia en forma teórica y se hallarán los valores numéricos en forma experimental.

Retornando a la pregunta debemos indicar que la eficiencia se mide en tiempo. Para esto diremos que un programa toma un tiempo del orden $t(n)$ para una instancia de tamaño n. Más adelante trataremos un término más riguroso que es la notación asintótica.

Si tratamos el tiempo en segundos uno podría pensar si tengo una máquina más rápida el tiempo será menor. ¿Cómo es que este método teórico llega a ser efectivo?

Consideremos que el algoritmo toma un tiempo en segundos $t_1(n)$ para una máquina en particular y $t_2(n)$ para otra. Para un número n suficientemente grande podemos hallar unas constantes a y b tales que $at(n) = bt(n)$ esto nos dice que cualquier ejecución del algoritmo está acotado por una función $t(n)$ y existen unas constantes positivas de proporcionalidad que nos dan el tiempo. Esto indica que si cambiamos de equipo podemos ejecutar el algoritmo 10 o 100 veces más rápido y que este incremento está dado solo por una constante de proporcionalidad.

La notación asintótica dice que un algoritmo toma un tiempo *del orden de*, en función del tamaño de la instancia . Existen algoritmos que ocurren muy frecuentemente y ameritan tener un nombre que a su vez se denominan tasas de crecimiento.

Se denominan algoritmos:

- Constantes los que cuyo tiempo de proceso no depende del tamaño de la instancia.

- Lineales a los que toman tiempos proporcionales a n.

Figura 1.1: Ordenes de magnitud de algoritmos

- Cuadráticos a los que toman tiempos proporcionales a n^2.
- Cúbicos a los que toman tiempos proporcionales a n^3.
- Polinomiales o exponenciales a los que toman tiempos proporcionales a n^3, n^k.
- Logarítmicos los proporcionales a $\log n$.
- Exponenciales los que toman tiempos proporcionales a 2^n.

Aún cuando es deseable siempre un algoritmo más eficiente desde el punto de vista teórico podemos en muchos casos escoger en la implementación otro, dado que las constantes ocultas pueden ser muy grandes y no llegar a ser tan eficientes para las instancias que deseamos procesar. La figura 1.1 representa las ordenes de magnitud de los algoritmos.

1.5. Qué considerar y qué contar

Consideremos por ejemplo que queremos hallar el máximo de tres números a, b, c. Para resolver ésto escribimos el siguiente código:

```
maximo=0
maximo=Max(a,b)
maximo=Max(c,maximo)

Max (a,b)
    if a > b
       return a
    else
       return b
```

Se puede ver que se requieren exactamente dos comparaciones para determinar el valor máximo de a, b, c. Si escribiéramos Max(Max(a,b),c) también se realizarían dos comparaciones. Este mismo algoritmo lo podemos escribir como sigue:

```
maximo=0
if (a > b)
 if(a > c)
   maximo = a
 else
   maximo = c
else
   if b > c
     maximo = b
   else
     maximo = c
```

Vemos que aún cuando hemos codificado más comparaciones en la ejecución solo se ejecutan dos como el caso anterior. Se hace necesario establecer que es lo que debemos considerar en los programas, ésto hace necesario definir, lo que se entiende por operaciones elementales.

1.5.1. Operaciones elementales

Es una operación cuyo tiempo de ejecución está acotado por una constante que solo depende de una implementación particular como ser la máquina, el lenguaje de programación, etc.

En el ejemplo que realizamos estaremos interesados en medir el número de comparaciones necesarias para hallar el máximo. El resto de las operaciones las consideramos operaciones elementales.

En el caso de la suma de elementos de un vector lo que deseamos medir es la cantidad de sumas realizadas para resolver el problema y las operaciones de comparación necesarias para implementar una solución, se consideran operaciones elementales.

1.5.2. Análisis del mejor caso, caso medio y peor caso

El tiempo que toma un algoritmo puede variar considerablemente en función de diferentes instancias de un mismo tamaño. Para comprender esto mejor consideremos el ejemplo del programa de clasificación por inserción.

```
inserción(t)
for i=1 to n
    x= t[i]
    j=i-1
    while j > 0 and x<t[j]
       t[j+1]=t[j]
       j=j-1
    t[j+1]=x
```

Consideremos un conjunto A en el cual queremos procesar el algoritmo de ordenar. Para ordenar A utilizaríamos *inserción(A)*. Si la instancia A está previamente ordenada en forma ascendente podemos probar que el tiempo de proceso es el óptimo porque solo inserta valores en el vector y no realiza intercambios.

En el caso de que el vector estuviera ordenado en orden descendente por cada elemento hay que recorrer todo el vector intercambiando los valores. Este claramente es el peor comportamiento que podemos tener.

Si el vector estuviera ordenado en forma ascendente no realizaría ningún intercambio y este es el mejor caso en el que se puede estar.

Sin embargo si el vector estuviera desordenado en forma aleatoria el comportamiento se considera del caso medio.

Cuando el tiempo de respuesta de un problema es crítico normalmente analizamos el peor caso que es mucho más fácil de analizar que el caso medio. Por lo tanto un análisis útil de un algoritmo requiere un conocimiento a priori de las instancias que vamos a procesar.

1.5.3. Ejercicios

1. Si deseamos verificar que un número es primo, explique si lo que se desea contar para hallar la eficiencia del algoritmo es: a) el número de dígitos b) el número de divisiones que se deben realizar.

2. Indique que es lo que deberíamos contar en el caso de escribir un algoritmo para multiplicar dos números a fin de hallar el número de operaciones requeridas para completar el algoritmo.

3. Suponiendo que el tiempo de proceso de una instancia es un mili segundo.. ¿En qué porcentaje se incrementa el tiempo de proceso al doblar el tamaño de la instancia ? si el algoritmo es a) exponencial, b) logarítmico, c) cuadrático?

1.5.4. Laboratorio

Se quiere analizar el comportamiento del algoritmo de ordenar por el método de inserción en forma experimental. Para esto seguiremos los siguientes pasos:

1. Codificar el algoritmo.

2. Crear una instancia ordenada en forma ascendente.

3. Tomar el tiempo de proceso para diferentes tamaños de instancias.

4. Repetir el proceso para una instancia ordenada en forma descendente.

5. Luego para una instancia generada aleatoriamente.

6. Tabular los datos.

7. Interpretar los resultados.

Para realizar éste experimento construimos el siguiente programa Java:

```
import java.util.*;

public class Isort {

    /**
     * Rutina Principal para la prueba experimental del
     * programa de ordenar por el metodo de insercion
     *
     * @author Jorge Teran
     * @param args
     *    0 crear las instancias en forma Descendente
     *    1 crear las instancias en forma Ascendente
     *    2 crear las instancias en forma Aleatoria
     */
    public static void main(String[] args) {
        //opcion por linea comando
        // int opcion = Integer.parseInt(args[0]);
        int opcion = 0; // parmetro por programa
        int instanciaMaxima = 1000000;
        long tiempoInicio, tiempoFin, tiempoTotal;
        for (int i = 10; i < instanciaMaxima; i = i * 10) {
            Arreglo arreglo = new Arreglo(opcion, i);
            tiempoInicio = System.currentTimeMillis();
            arreglo.Ordena();
            tiempoFin = System.currentTimeMillis();
            tiempoTotal = tiempoFin - tiempoInicio;
            System.out.println("Tamaño de la instancia " + i
                    + " Tiempo de proceso " + tiempoTotal);
        }
    }
}

class Arreglo {
    public int[] t = new int[1000000];

    Random generador = new Random();
```

```
int n;

Arreglo(int opcion, int n) {
    this.n = n;
    for (int i = 0; i < n; i++) {
        switch (opcion) {
        case 0:
            t[i] = i;
            break;
        case 1:
            t[i] = n - i;
            break;
        case 2:
            t[i] = generador.nextInt(10000);
            break;
        }

    }
    // listar(t);
}

void Ordena() {
    int x, j;
    for (int i = 1; i < n; i++) {
        x = t[i];
        j = i - 1;
        while ((j > 0) && (x < t[j])) {
            t[j + 1] = t[j];
            j = j - 1;

        }
        t[j + 1] = x;
    }
    // listar(t);
    return;
}
```

Tamaño Instancia	Ordenada Ascendente	Ordenada Descendente	instancia Aleatoria
10	0	0	0
100	0	1	1
1000	0	10	33
10000	1	163	124
100000	6	15812	7890

Cuadro 1.1: Tiempo de ejecución en milisegundos

```
    void listar(int[] t) {
        for (int i = 1; i < n; i++)
            System.out.print(t[i] + " ");
        System.out.println();
    }
}
```

El resultado de los tiempos de ejecución, en mili segundos, obtenidos en la prueba del programa se ven en el cuadro 1.1.

Como se ve el peor caso es cuando la instancia que creamos está ordenada en forma descendente. El caso medio se da cuando el vector está ordenado aleatoriamente. En diferentes ejecuciones del programa veremos que el tiempo de proceso que se obtiene en el caso medio varía. Esto se debe a que los datos que se generan no están en el mismo orden. Situación que no se da en las otras instancias generadas. Una buena aplicación de éste código se da en los casos en que los conjuntos de datos a los que vamos aplicar el método, están casi totalmente ordenados.

Capítulo 2

Análisis de algoritmos

2.1. Introducción

Para el análisis de los algoritmos es muy importante determinar la eficiencia de los mismos. Esto se logra a través del cálculo del tiempo de proceso o complejidad . Como no existen mecanismos normalizados para el cálculo de estos tiempos, solo nos referimos al tiempo tomado por el algoritmo multiplicado por una constante. A esto denominamos notación asintótica. Existen tres conceptos, la notación omega, la notación teta y la notación O grande simbolizadas por Ω, θ y O respectivamente. En el texto se enfocará al análisis del tiempo de proceso con la notación O. Se recomienda la bibliografía [McC01] y [EH96]

2.2. Notación orden de

La notación *orden de* nos permite especificar la magnitud máxima de una expresión. Sea por ejemplo una función $t(n) = 5n^2$ y podemos pensar que n es como el tamaño de una instancia de un algoritmo dado. Ahora supongamos que esta función representa la cantidad de recursos que gasta el algoritmo, ya sea el tiempo de proceso, almacenamiento de memoria u otros. Las constantes solo representan valores de una implementación dada y no tienen que ver con el tamaño de la instancia.

En este caso decimos que esta función $t(n)$ es del orden n^2 dado que solo está acotado por una constante. Esto se denomina *O grande de t(n)* y matemáticamente lo representamos como $O(n^2)$.

2.2.1. Propiedades de O grande de n

Para aclarar lo que representa *O grande de n* describamos sus propiedades:

1. Regla del valor máximo. Dadas dos funciones $f(n)$ y $g(n)$ que pertenecen a los números reales positivos incluyendo el cero

   $$O(f(n) + g(n)) = max(f(n), g(n))$$

   por ejemplo si se tiene la función $t(n) = 5n^3 + \log n - n$ aplicando la regla del valor máximo tenemos que $O(t(n)) = max(5n^3, \log n, -n)$ cuyo valor es $O(n^3)$.

2. Igualdad de logaritmos. En la notación O la expresión

   $$O(\log_a n) = O(\log_b n)$$

   que, claramente es incorrecto en las matemáticas, pero es correcta en la notación O.

   Esto se demuestra aplicando la propiedad de los logaritmos para el cambio de base $\log_a n = \log_b n / log_b a$. Notamos que el denominador es una constante y como en la notación O no se escriben las constantes queda demostrado.

3. Propiedad reflexiva. La notación ϵO es reflexiva dado que:

   $$f(n)\epsilon O(f(n))$$

4. Propiedad transitiva.

   $$\text{Si} f(n)\epsilon O(g(n)) \text{y} g(n)\epsilon O(h(n)) \text{entonces} f(n)\epsilon O(h(n)).$$

5. Propiedades de límites.

   a) si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \epsilon \Re^+ \Rightarrow f(n)\epsilon O(g(n))$ y $g(n)\epsilon O(f(n))$

   b) si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n)\epsilon O(g(n))$ pero $g(n)\neg\epsilon O(f(n))$

   c) si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow$
   $f(n)\neg\epsilon O(g(n))$ pero $g(n)\epsilon O(f(n))$

2.3. Notación asintótica condicional

Muchos algoritmos son más fáciles de analizar si restringimos nuestra atención a instancias que cumplen alguna condición tales como ser una potencia de 2, ser de un tamaño determinado. Consideremos por ejemplo, un algoritmo cuyo resultado para $n = 1$ es $t(n) = 1$ y para otros valores toma $nt(n-1)$. Este algoritmo se expresa como una solución recursiva que se escribe como sigue:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ nT(n-1) & \text{en otros casos.} \end{cases}$$

en las secciones siguientes estudiamos como resolver éstas ecuaciones denominadas recurrencias.

2.4. Notación omega

La notación omega que la representamos por Ω se utiliza para representar la cota inferior de un algoritmo. Por ejemplo el algoritmo de ordenamiento por inserción demora $O(n^2)$ en el peor caso sin embargo la cota inferior para los programas de clasificación que trabajan únicamente por comparaciones es $\Omega(n \log n)$ por lo tanto podemos decir que no existen algoritmos de clasificación que en su peor caso sean mejores que $\Omega(n \log n)$.

Sea $t(n) \epsilon \Omega(f(n))$ si existe una constante real y positiva d tal que la relación $t(n) \geq df(n)$ sea cierta para una cantidad infinita de valores de n, si buscamos que la relación se cumpla para un número finito de valores de n, entonces $f(n)$ es el límite inferior del algoritmo.

2.5. Notación teta

Cuando analizamos un algoritmo estaríamos muy satisfechos si nuestro algoritmo estuviera acotado simultáneamente por Ω y O. Por esta razón se introduce la notación θ. Decimos que un algoritmo es en $\theta(f(n))$ o que $f(n)$ es de orden exacto si:

$$\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

Esta notación se utiliza muy poco y como podrá apreciar tiene un mayor grado de dificultad. Comunmente se utiliza la notación O y para mostrar la

relación contra el mejor algoritmo posible, para el cual, usamos la notación Ω.

2.6. Análisis de las estructuras de control

2.6.1. Secuenciales

Sean p_1 y p_2 dos fragmentos de un algoritmo y sean t_1 y t_2 los tiempos que toman cada uno respectivamente. La regla de secuencias indica que el tiempo de ejecución de la secuencia $p_1; p_2$ toma un tiempo $t_1 + t_2$. Por la regla del valor máximo el tiempo es $(max(O(t_1), O(t_2))$. Este análisis considera que p_1 y p_2 son independientes. Esto es que, el proceso p_2 no depende de los resultados de p_1.

2.6.2. Ciclos For

Consideremos el programa siguiente

```
for i = 1 to m
    p(i)
```

En este programa no confundamos m con n que es el tamaño de una instancia. Si $p(i)$ toma un tiempo constante t y no depende de i entonces este tiempo se repetirá m veces y denotamos esto como:

$$\sum_{i=1}^{m} t = t \sum_{i=1}^{m} 1 = tm = O(m)$$

En un caso mas general donde $t(i)$ no es constante y depende de i deberíamos calcular el resultado de la siguiente sumatoria:

$$\sum_{i=1}^{m} t(i)$$

2.6.3. Llamadas recursivas

El análisis recursivo de un algoritmo es normalmente casi directo y una simple inspección normalmente nos da una ecuación de recurrencia que representa el comportamiento del mismo. Consideremos el problema de buscar un elemento t en un arreglo a recursivamente

```
busca (n)
   if n =0  return -1
   else
   if a[n]= t
   return n
   else
   return busca(n-1)
```

En este algoritmo podemos ver que para los casos en que n es 0 el tiempo que toma es una constante, cuando encuentra el valor también el tiempo es una constante. En los otros casos depende del valor $n-1$. Esto podemos representar por una función recurrente de la siguiente forma:

$$T(n) = \begin{cases} 1 & \text{si } n = 0, \\ 1 & \text{si } a[n] = t, \\ T(n-1) + h(n) & \text{en otros casos.} \end{cases}$$

El valor de h(n) es el número de operaciones elementales requeridas en cada recursión. Podemos resolver esta recurrencia y demostrar que la solución es O(n).

2.6.4. Ciclos while y repeat

Los ciclos while y repeat son mucho más difíciles de analizar porque no existe una forma de saber cuantas veces se va a ejecutar. Para terminar un ciclo hay que probar alguna condición que nos dará un valor de terminación.Consideremos el código siguiente :

```
ejemplo(n)
   i=0
   while n>1
      if par(n)
         n=n/2
         i=i+1
      else
        n=3n/2
        i=i+1
return i
```

En este ejemplo no podemos decidir a simple vista cuantas veces se ejecuta la instrucción $i = i + 1$ se ve que cada vez que n es una potencia de 2 se pasará por la parte $par(n)$ y cuya cantidad de veces será el $\log_2 n$. ¿Qué podemos decir cuando es impar?. ¿El algoritmo termina?. Estas interrogantes las dejamos para el análisis del lector.

En muchos casos será necesario aplicar la teoría de probabilidades para poder determinar las veces que una instrucción se ejecuta. Si podemos deducir una recurrencia a partir del algoritmo podemos hallar fácilmente su $O(n)$.

```
ejemplo(n)
   i=0
   while n>1
         n=n/2
         i=i+1
return i
```

En este ejercicio vemos que los valores que toma n son $n, n/2, n/4...$ por lo tanto podemos escribir esta recurrencia como

$$T(n) = \begin{cases} 1 & \text{si } n < 2, \\ T(n/2) + 1 & \text{en otros casos.} \end{cases}$$

y luego de resolverla podemos indicar que el resultado es $O(\log n)$. Los métodos para resolver recurrencias se explicarán más adelante.

2.6.5. Análisis del caso medio

Consideremos el algoritmo de ordenar por el método de inserción

```
inserción(t)
for i=1 to n
    x= t[i]
    j=i-1
    while j > 0 and x<t[j]
       t[j+1]=t[j]
       j=j-1
    t[j+1]=x
```

De este algoritmo podemos ver que en el mejor caso, vale decir cuando los datos vienen ordenados en forma ascendente, el tiempo es lineal y el peor

caso es de orden cuadrático que se da en el caso inverso.¿Cuál es el comportamiento en el caso promedio? Para resolver esta interrogante se hace necesario conocer a priori la función de distribución de probabilidad de las instancias que, el algoritmo debe resolver. La conclusión del análisis del caso medio depende de esta suposición.

Supongamos que todas las permutaciones posibles de las instancias pueden ocurrir con la misma probabilidad para determinar el resultado en promedio debemos sumar el tiempo que, toma ordenar cada uno de las $n!$ posibles instancias. Definamos el rango parcial de $T[i]$ como la posición que, ocuparía en el arreglo si estuviera ordenado. Supongamos que $1 \leq i \geq n$ y estamos ingresando en ciclo while, $T[i-1]$ contiene los mismos elementos que antes, dado que $T[i]$ todavía no se ha insertado. Este rango parcial estará entre 1 y i y lo llamaremos k.

La cantidad de veces que los elementos del vector se desplazarán es $i - k + 1$ porque los elementos hasta $i - 1$ ya están ordenados. Como tenemos i elementos la probabilidad que i ocurra es $1/i$ de aquí calculamos el tiempo que toma el ciclo while

$$c_i = \frac{1}{i}\sum_{k=1}^{i}(i-k+1) = \frac{i+1}{2}$$

como tenemos n elementos en el ciclo principal debemos sumar los tiempos de las instrucciones interiores dando

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n}\frac{i+1}{2} = \sum_{i=1}^{n}\frac{i}{2} + \sum_{i=1}^{n}\frac{1}{2} = \frac{n^2+3n}{4}$$

2.6.6. Análisis amortizado

El análisis del peor caso es muchas veces muy pesimista. Supongamos que tenemos un programa con un vector que almacena valores y queremos encontrar el valor mínimo. Si el vector está ordenado devolver el primero toma un tiempo constante. Supongamos que enviamos a esta rutina un número y si existe devuelve el valor mínimo y si no existe inserta el valor y luego devuelve el valor mínimo. ¿Cuánto tiempo toma el algoritmo? En el peor caso toma el tiempo de insertar y de buscar, sin embargo muchas veces solamente tomará el tiempo de buscar. Para entender el tiempo amortizado de este algoritmo podemos pensar que, en cada búsqueda ahorramos un poco

para gastarlo cuando tengamos que realizar un inserción. Esto nos muestra claramente que el tiempo amortizado depende de la probabilidad de que tengamos inserciones y cual la probabilidad de que tengamos solo búsquedas. La esperanza matemática es una buena medida para estos análisis.

2.7. Solución de recurrencias

Como ya vimos los algoritmos se pueden representar en forma de ecuaciones recurrentes aquí explicaremos tres métodos de solución, substitución o adivinación, iterativos y un métdo general para casos especiales.

2.7.1. Método por substitución

El método de substitución que bien puede llamarse método por adivinación se basa en adivinar de que forma es la solución, luego aplicamos la hipótesis inductiva para valores menores y demostrando que ésto se cumple, decimos que nuestra suposición es correcta. Consideremos el siguiente ejemplo:

$$T(n) = \begin{cases} 1 & n = 1, \\ 2T(n/2) + n & \text{otros casos.} \end{cases} \tag{2.7.1}$$

La recurrencia 2.7.1 puede escribirse como la ecuación

$$2T(n/2) + n \tag{2.7.2}$$

Primero buscamos una solución y suponemos que el tiempo asintótico $O(n)$ de esta ecuación es $O(n \log n)$. Debemos probar que ésta es una función para la cual todos los valores de $T(n)$ son menores con lo cual se demuestra que es una cota superior.

Suponiendo que esta solución es aplicable a $n/2$ para una constante arbitraria c en la ecuación 2.7.2 tenemos:

$$\begin{aligned} T(n) &\leq 2c(\frac{n}{2} \log \frac{n}{2}) + n \\ &= cn \log \frac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq n \log n \text{ para } c \geq 1 \end{aligned}$$

Con lo cual estaría probada nuestra suposición. Como verá en algunas condiciones es difícil de probar esta recursión. Por ejemplo, consideremos que $T(1) = 1$ reemplazando nuestra hipótesis se tiene $T(1) = c1 \log 1$ que da $T(1) = c0$ y para cualquier valor de c vemos que no es posible probar nuestra suposición por lo que, es usual imponer algunas restricciones tales como probar desde $T(2)$.

Realizar suposiciones acertadas es muy complicado, por lo que se hace necesario realizar diferentes pruebas con valores máximos y mínimos para hallar la solución. Lo que estamos realizando es una hipótesis inductiva y en muchas ocasiones podemos agregar más constantes por ejemplo, $T(n) \leq cn - b$, el restar algunas constantes de los términos de menor orden es considerada por muchos como una solución intuitiva.

Este método es un problema, la cota superior probada no garantiza que existe otra cota superior que sea menor a la probada. Consideremos por ejemplo que se hubiese elegido n^2 como una cota superior también se hubiera probado cierta porque $O(n \log(n)) \in O(n^2)$.

2.7.2. Cambio de variables

En muchos casos las soluciones tienen formas que no hemos visto antes por lo que no podemos plantear la solución. Consideremos la ecuación:

$$T(n) = 2T(\sqrt{n}) + \log n$$

Podemos simplificar esta ecuación reemplazando n por 2^m que nos da

$$T(2^m) = 2T(2^{\frac{m}{2}}) + \log 2^m$$

$$T(2^m) = 2T(2^{\frac{m}{2}}) + m$$

ahora reemplacemos $T(2^m)$ por $S(m)$

$$S(m) = 2S(\frac{m}{2}) + m$$

Esta solución ya la conocemos y sabemos que es $O(n \log n)$ entonces reemplazando en esta respuesta las substituciones que hemos realizado

$$T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n).$$

2.7.3. Ejercicios

1. Muestre que la solución de $T(n) = T(n/2) + 1$ es $O(\log n)$
2. Muestre que la solución de $T(n) = T(n/2 + 17) + 1$ es $O(\log n)$
3. Resuelva la recurrencia $T(n) = T(\sqrt{n}) + 1$ realizando cambio de variables

2.7.4. Método iterativo

El método iterativo es mucho mejor porque no requiere adivinanzas que, pueden hacernos suponer verdaderos resultados erróneos. Pero a cambio se requiere un poco más de álgebra. Consideremos la siguiente recursión:

$$T(n) = \begin{cases} 1 & n = 1, \\ T(\frac{n}{2}) + 1 & \text{otros casos.} \end{cases}$$

para resolver ésta vamos comenzar en $T(n)$ y vamos a desarrollar la misma

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \text{ reemplazando n por } \frac{n}{2} \\ &= T(\frac{n}{4}) + 2 \\ &= T(\frac{n}{8}) + 3 \end{aligned}$$

hasta hallar la forma general

$$= T(\frac{n}{2^k}) + k$$

cuando $n = 1$ termina y por lo tanto $T(1)$ toma la forma de:

$$T(1) = 1 = \frac{n}{2^k}$$

reemplazando tenemos:

$$T(n) = 1 + k = 1 + \log n$$

de aquí sabemos que el tiempo asintótico de esta recursión es $O(\log n)$. Resolvamos con este método la ecuación 2.7.1

$$T(n) = \begin{cases} 1 & n = 1, \\ 2T(\frac{n}{2}) + n & \text{otros casos.} \end{cases}$$

desarrollando la recursión tenemos

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\ &= 4T(\frac{n}{4}) + 2\frac{n}{2} + n \\ &= 4T(\frac{n}{4}) + 2n \\ &= 4(2T(\frac{n}{8}) + \frac{n}{4}) + 2n \\ &= 8T(\frac{n}{8}) + 3n \text{ termina cuando} \\ &= 2^k T(\frac{n}{2^k})) + kn \\ &= 2^k T(1) + kn \end{aligned}$$

Este algoritmo termina cuando llegamos a $T(1)$; donde toma el valor de 1 cuando $n/2^k = 1$. Despejando el valor de k tenemos $k = \log n$ reemplazando:

$$\begin{aligned} T(n) &= 2^k + kn \\ &= n + n\log n \\ &= O(n\log n) \end{aligned}$$

Ejemplo:
Dar las cotas superiores para T(n) en las siguiente recurrencia. Asumiendo que T(n) es constante para n $\leq$ 1.

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

Solución:
Desarrollando la recursión tenemos:

$$\begin{aligned} T(n) &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &= T(n-(n-1)) + (n-(n-2)) + ... + (n-1) + n \end{aligned}$$

que toma la forma de

$$\sum_{i=1}^{n} i$$

resolviendo tenemos

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

por lo que la cota superior es $O(n^2)$

2.7.5. Ejercicios

1. Resolver $T(n) = 3T(n/2) + n$.
2. Resolver $T(n) = T(n/3) + T(2n/3) + n$.
3. Resolver $T(n) = T(n-a) + T(a) + n$

2.7.6. Teorema master

Existen muchas recursiones que tienen la forma

$$T(n) = aT(\frac{n}{b}) + f(n) \tag{2.7.3}$$

La ecuación 2.7.3 es un caso típico de soluciones que aplican la técnica *divide y vencerás* donde a representa el número de llamadas recursivas $T(n/b)$ el tamaño de las llamadas y $f(n)$ el trabajo realizado en cada recursión. Esta ecuación nos da las siguientes soluciones:

$$T(n) = \begin{cases} \text{si } f(n) = n^{\log_b a - \epsilon}, \epsilon \geq 0 & \Rightarrow O(n^{\log_b a}), \\ \text{si } f(n) = n^{\log_b a} & \Rightarrow O(n^{\log_b a} \log n), \\ \text{si } f(n) = n^{\log_b a + \epsilon}, \epsilon \geq 0 & \Rightarrow O(f(n)), \\ \text{para polinomios donde} & \\ f(n) = cn^k \text{ se simplifica} & \\ \text{si } a > b^k & \Rightarrow O(n^{\log_b a}), \\ \text{si } a = b^k & \Rightarrow O(n^k \log n), \\ \text{si } a < b^k & \Rightarrow O(n^k) \end{cases} \tag{2.7.4}$$

La demostración del teorema se puede consultar en la bibliografía [GB96] citada al final del texto.

Resuelva los siguientes ejercicios utilizando el teorema master:

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

En este caso vea que $a = 2, b = 2, k = 1$ y $f(n) = n^k$ aplicando el teorema estamos en el caso $a = b^k$ por lo que la soluciones es $T(n) = O(n^k \log n) = O(n \log n)$

2.
$$T(n) = 9T(n/3) + n$$

Veamos $a = 9, b = 3, k = 1$ por lo que $a \geq b^k$ entonces la solución es $n^{\log_3 9} = O(n^2)$

3.
$$T(n) = T(2n/3) + 1$$

Veamos $a = 1, b = 3/2, k = 0$ por lo que $a = b^k$ entonces la solución es $O(n^k \log n) = O(\log n)$

Ejercicios

Resolver utilizando el teorema master.

1. $T(n) = 4T(n/2) + n$.
2. $T(n) = 4T(n/2) + n^2$.
3. $T(n) = 4T(n/2) + n^3$.

2.7.7. Solución general de recurrencias lineales

Las recurrencias de la forma

$$a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = f(n) \tag{2.7.5}$$

se denominan: ecuación lineal, homogénea y de coeficientes constantes. Cuando $f(n)$ es 0 se denomina homogénea y en otros casos no homogénea. Una buena descripción de como resolver estas recurrencias pueden ver en los libros de Grimaldi [Gri77] y Knuth [RLG94]

Solución de recurrencias homogéneas

La solución de una ecuación $t(n)$ puede representarse como la combinacion lineal de funciones por ejemplo $c_1 f(n) + c_2 g(n)$.Consideremos la expresión

$$p(x) = a_0 x^k + a_1 x^{k-1} + ... + a_k = 0$$

La solución trivial es cuando $x = 0$ no es de interés. Esta ecuación se denomina ecuación característica o polinomio característico. Recordando el álgebra, un polinomio de orden k tiene exactamente k raíces y puede factorizarse como

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

donde r_i son las soluciones de $p(x) = 0$. Cualquier r_i es una solución a éste polinomio por lo que también son una solución a $t(n)$ porque cualquier combinacion lineal tambien conforma una solución, por lo que podemos concluir que

$$t(n) = \sum_{i=1}^{k} (c_i r_i^n)$$

las soluciones se dan para cualquier constante que se escoja, siempre que las raíces sean distintas.

Ejemplo

Consideremos la secuencia de Fibonacci:

$$f(n) = \begin{cases} n & \text{si } n = 0, n = 1, \\ f(n-1) + f(n-2) & \text{otros casos.} \end{cases}$$

reescribimos esta ecuación para satisfacer la ecuación 2.7.5

$$f(n) - f(n-1) - f(n-2) = 0$$

Esta ecuación tiene un polinomio característico

$$x^2 - x - 1 = 0$$

que tiene como solución las raíces

$$r_1 = \frac{1 + \sqrt{5}}{2} \text{ y } r_2 = \frac{1 - \sqrt{5}}{2}$$

entonces la solución general es

$$f(n) = c_1 r_1^n + c_2 r_2^n$$

ahora utilizando las condiciones iniciales podemos hallar las constantes c_1, c_2

$$c_1 + c_2 = 0$$
$$r_1 c_1 + r_2 c_2 = 1$$

resolviendo obtenemos

$$c_1 = \frac{1}{\sqrt{5}} \text{ y } c_2 = -\frac{1}{\sqrt{5}}$$

reemplazando en la ecuación obtenemos

$$f(n) = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$$

Solucición de recurrencias no homogéneas

La recurrencias de la forma

$$a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = b^n p(n) \tag{2.7.6}$$

- b es una constante
- $p(n)$ es un polinomio en n de grado d.

Consideremos la recurrencia siguiente:

$$t(n) - 2t(n-1) = 3^n \tag{2.7.7}$$

en este caso $b = 3$ y $p(n) = 1$ es un polinomio de grado 0, para resolver este problema primero lo convertimos a un problema familiar, la recurrencia homogénea. Multiplicamos por tres la ecuación 2.7.7

$$3t(n) - 6t(n-1) = 3^{n+1} \tag{2.7.8}$$

ahora cambiemos $n-1$ por n

$$3t(n-1) - 6t(n-2) = 3^n \tag{2.7.9}$$

restando 2.7.7 de 2.7.8 tenemos una ecuación homogénea

$$t(n) - 5t(n-2) + 6t(n-2) = 0 \tag{2.7.10}$$

el polinomio característico es

$$x^2 - 5x + 6 = (x-2)(x-3)$$

por lo que, las soluciones son de la forma

$$t(n) = c_1 2^n + c_2 3^n$$

Sin embargo no es cierto que cualquier constante arbitraria producirá la solución correcta porque la ecuación 2.7.7 no es la misma que la 2.7.10. Podemos resolver la ecuación original en base de $t(0)$ y $t(1)$. La función original implica que $t_1 = 2t_0 + 3$ por lo que, las ecuaciones a resolver son:

$$c_1 + c_2 = t(0)$$
$$2c_1 + 3c_2 = 2t(0) + 3$$

de donde obtenemos $c_1 = t(0) - 3$ y $c_2 = 3$. Resolviendo esto tenemos

$$t(n) = (t(0) - 3)2^n + 3^{n+1}$$

que es independiente de la solución inicial $t(n) \epsilon O(3^n)$

2.8. Ejercicios resueltos

1. ¿Cuánto tiempo toma el siguiente algoritmo?

```
l=0
for i=1 to n
   for j=1 to n^2
      for k=1 to n^3
         l=l+1
```

Solución:

Recordando que los ciclos for son las sumas de los tiempos individuales de los procedimientos interiores podemos escribir

$$\begin{aligned} T(n) &= \sum_{i=1}^{n}(\sum_{j=1}^{n^2}(\sum_{k=1}^{n^3} 1)) \\ &= \sum_{i=1}^{n}(\sum_{j=1}^{n^2} n^3) \\ &= n^3 \sum_{i=1}^{n} n^2 \\ &= n^3 n^2 n = n^6 \end{aligned}$$

2. ¿Cuánto tiempo toma el siguiente algoritmo?

```
l=0
for i=1 to n
   for j=1 to i
      for k=1 to j
         l=l+1
```

Solución:

En este caso no todas las sumas son hasta n

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n}(\sum_{j=1}^{i}(\sum_{k=1}^{j}1)) \\
&= \sum_{i=1}^{n}(\sum_{j=1}^{i}j) \\
&= \sum_{i=1}^{n}i\frac{i+1}{2} \\
&= \sum_{i=1}^{n}(\frac{i^2}{2}+\frac{i}{2}) \\
&= \sum_{i=1}^{n}\frac{i^2}{2}+\sum_{i=1}^{n}\frac{i}{2} \\
&= \frac{n(n+1)(2n+1)}{12}+\frac{1}{2}n\frac{n+1}{2} \\
&= \frac{(n^2+n)(2n+1)}{12}+\frac{n+n^2}{4} \\
&= \frac{2n^3+3n^2+n}{12}+\frac{n+n^2}{4} \\
&= O(n^3)
\end{aligned}$$

3. Resolver utilizando el teorema master

```
procedure DC(n)
if n < = 1
      DC(n/2)
          for j=1 to n^3
          x[j]=0
```

Solución

Del análisis vemos que

$$T(n) = \begin{cases} 1 & n = 1, \\ T(n/2)+n^3 & \text{otros casos.} \end{cases}$$

De donde tenemos $a = 1, b = 2, k = 3$ lo que nos lleva al caso $1 < 2^3$ dando como solución $O(n^k) = O(n^3)$.

4. Resolver utilizando el teorema master

```
procedure DC(n)
if n < = 1
  for i=1 to 8
      DC(n/2)
  for j=1 to n^3
  x[j]=0
```

Solución

Del análisis vemos que hay la parte de algoritmo que divide el problema $DC(n/2)$ se procesa 8 veces lo que cambia el problema anterior a

$$T(n) = \begin{cases} 1 & n = 1, \\ 8T(n/2) + n^3 & \text{otros casos.} \end{cases}$$

De donde tenemos $a = 8, b = 2, k = 3$ lo que nos lleva al caso $8 = 2^3$ dando como solución $O(n^k \log n) = O(n^3 \log n)$.

5. Dar las cotas superiores para T(n) en las siguientes recurrencias. Asumiendo que T(n) es constante para n $\leq$ 1.

$$T(n) = 2T(\frac{n}{2}) + n^3$$

Solución:
Desarrollando la recursión tenemos:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n^3 \\ &= 2(2T(\frac{n}{4}) + (\frac{n}{2})^3) + n^3 \\ &= 4T(\frac{n}{4}) + 2(\frac{n^3}{2}) + n^3 \\ &= 4(2T(\frac{n}{8}) + (\frac{n}{4}))^3 + 2(\frac{n^3}{2}) + n^3 \\ &= 8T(\frac{n}{8}) + 4(\frac{n}{4})^3 + 2(\frac{n^3}{2}) + n^3 \end{aligned}$$

De aquí vemos que para un término i toma la forma

$$T(n) = 2^i T(\frac{n}{2^i}) + \sum_{k=0}^{i-1} 2^k (\frac{n}{2^k})^3$$

T(1) se da cuando $\frac{n}{2^i} = 1$ entonces $i = \log(n)$
reemplazando tenemos:

$$\begin{aligned} T(n) &= n + n^3 \sum_{k=1}^{i-1} (\frac{1}{2^k})^2 \\ &= n + n^3 \sum_{k=1}^{i-1} (\frac{1}{4^k}) \end{aligned}$$

$$\begin{aligned} \sum_{k=1}^{i-1} (\frac{1}{4^k}) &= \frac{1 - (\frac{1}{4})^{i-1}}{1 - \frac{1}{4}} \\ &= \frac{4}{3}(1 - \frac{1}{4^{i-1}}) \\ &= \frac{4}{3} - \frac{4}{3}\frac{1}{4^{i-1}} \\ &= \frac{4}{3} - \frac{16}{3}\frac{1}{2^{2i}} \\ &= \frac{4}{3} - \frac{16}{3}\frac{1}{4^{\log(n)}} \\ &= \frac{4}{3} - \frac{16}{3}\frac{1}{n^2} \\ T(n) &= n + n^3(\frac{4}{3} - \frac{6}{3}\frac{1}{n^2}) \\ &= n + n^3\frac{4}{3} - \frac{4}{3}n \end{aligned}$$

por lo que, la cota superior es $O(n^3)$

6. Dar las cotas superiores para T(n) en las siguientes recurrencias. Asumiendo que T(n) es constante para n $\leq$ 2.

$$T(n) = 2T(\frac{n}{4}) + \sqrt{n}$$

Solución:
Desarrollando la recursión tenemos:

$$\begin{aligned} T(n) &= 2(2T(\frac{n}{4^2}) + \sqrt{\frac{n}{4}}) + \sqrt{n} \\ &= 4T(\frac{n}{4^2}) + 2\sqrt{\frac{n}{4}} + \sqrt{n} \end{aligned}$$

De aquí vemos que para un término i toma la forma

$$T(n) = 2^i T(\frac{n}{4^i}) + \sum_{k=1}^{i-1} 2^k (\sqrt{\frac{n}{4^k}})$$

t(1) se da cuando $\frac{n}{4^i} = 1$ entonces $i = \log_4(n)$ además notamos que $4^i = (2^i)^2$ reemplazando tenemos:

$$\begin{aligned}
&= 2^i T(1) + \sum_{k=1}^{i-1} 2^k \sqrt{\frac{n}{4^k}} + \sqrt{n} \\
&= \frac{n}{2} + \sqrt{n} + \sum_{k=1}^{i-1} 2^k \sqrt{\frac{n}{2^{k^2}}} \\
&= \frac{n}{2} + \sqrt{n} + \sum_{k=1}^{i-1} \sqrt{n} \\
&= \frac{n}{2} + \sqrt{n} + (i-1)\sqrt{n} \\
&= \frac{n}{2} + \sqrt{n} + (\log_4(n) - 1)\sqrt{n} \\
&= \frac{n}{2} + \sqrt{n} + \log_4(n)\sqrt{n} - \sqrt{n}
\end{aligned}$$

por lo que la cota superior es $O(\log(n)\sqrt{n})$.

7. Dar las cotas superiores para T(n) asumiendo que T(n) es constante para n $\leq$ 2.

$$T(n) = 2T(\sqrt{n}) + 1$$

Solución:
Desarrollando la recursión tenemos:

$$\begin{aligned}
T(n) &= 2T(n^{\frac{1}{2}}) + 1 \\
&= 4T(n^{(\frac{1}{2})^2}) + 2 + 1 \\
&= 8T(n^{(\frac{1}{2})^3}) + 4 + 2 + 1
\end{aligned}$$

que toma la forma de

$$\begin{aligned}
T(n) &= 2^i T(n^{(\frac{1}{2})^i}) + \sum_{k=0}^{k=i-1} 2^k \\
&= 2^i T(n^{(\frac{1}{2})^i}) + 2^i - 1
\end{aligned}$$

cuando $n^{(\frac{1}{2})^i} = 2$ estamos en T(2) que es constante

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

$$2^i = \log(n)$$

$$T(n) = \log(n)T(2) + \log(n) - 1 = 2\log(n) - 1$$

la cota superior es $O(\log n)$

2.9. Ejercicios

1. Resolver las siguientes recurrencias utilizando el teorema master, verificar con el método de substitución y desarrollando la recurrencia. Asuma que T(1)=1.

   a) $T(n) = 2T(\frac{n}{2}) + \log(n)$
   b) $T(n) = 3T(\frac{n}{3}) + \log(n)$
   c) $T(n) = 4T(\frac{n}{4}) + \log(n)$
   d) $T(n) = 2T(\frac{n}{2}) + n^2$
   e) $T(n) = 2T(\frac{n}{2}) + n^3$
   f) $T(n) = 2T(\frac{n}{2}) + n^4$
   g) $T(n) = \sqrt{(}n)T(\sqrt{n}) + n$

2. Hallar el $O(n)$ del siguiente programa de búsqueda binaria.

```
int l,u,m;
l=0; u=n-1;
while(l<u) {
   m=(l+u)/2
   if (a[mid]< x)
     l=m+1
   else
   if (a[mid]> x)
       l=m-1
   else
```

```
   return ("hallado")
   }
   return ("no existe">)
```

3. Hallar el $O(n)$ del siguiente programa para hallar el máximo común divisor. Sugerencia escribir el tamaño de la instancia en cada iteración y aproximar a una serie armónica.

```
mcd(m,n) {
   while (n> 0){
      resto=m%n
      m=n
      n=resto
     }
     return (m)
 }
```

4. Hallar el $O(n)$ del siguiente programa de exponenciación.

```
pot(x,n) {
if (n==0)
   return 1
if (n==1)
   return x
if (par(n))
   return (pot(x*x,n/2))
else
   return (pot(x*x,n/2)*x)
}
```

5. Realizar una prueba experimental para comprobar las cotas superiores de los programas anteriormente propuestos.

6. Codificar el programa de búsqueda binaria en forma recursiva y hallar el $O(n)$

7. Se tienen dos programas A y B cuyo tiempo de ejecución es $150\log(n)$ milisegundos y $150n^2$ respectivamente. Indique que programa es mejor para $n < 100$,$n < 1,000$, $n > 10,000$

   ¿Como es el comportamiento si $150\log(n)$ cambia por $2000\log(n)$.

2.10. Nota sobre el cálculo integral

Cuando tenemos una sumatoria, desde nuestras clases de cálculo, asociamos la misma a una integral. Como la sumatoria es acotada, se podría pensar en utilizar una integral definida.

Consideremos por ejemplo la siguiente sumatoria y su resultado:

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{2.10.1}$$

Si suponemos que es posible hallar el resultado integrando la sumatoria se tiene lo siguiente:

$$\int i = 0ni^2 = \frac{i^3}{3}|_0^n = \frac{n^3}{3} + c \tag{2.10.2}$$

Claramente las ecuaciones 2.10.1 y 2.10.2 son diferentes. ¿Entonces para qué podemos utilizar el cálculo integral?

Bien, si queremos obtener la cota superior en la ejecución de un algoritmo, debemos obtener $o(f(n))$ de las 2.10.1 y 2.10.2

$$O(\frac{n(n+1)(2n+1)}{6}) = n^3$$

$$O(\frac{n^3}{3} + c) = n^3$$

Como ve, ambos resultados nos dan la misma cota superior. En el libro de matemáticas concretas de Donald Knuth [RLG94] se puede encontrar un método para hallar el resultado de una suma a través del cálculo.

Capítulo 3

Teoría de números

3.1. Introducción

Se hace muy importante la teoría de números en el aprendizaje de la programación. Este conocimiento permite comprender los aspectos que hacen al proceso computacional, tales como las capacidades de la máquina, el tamaño máximo de números que se pueden tratar y como trabajar con números más grandes.

En el tratamiento de este capítulo se toman en cuenta los aspectos relativos al tiempo de proceso para hacer eficiente los algoritmos expuestos.

3.2. Variables del lenguaje Java

El lenguaje de programación Java tiene varios tipos de variables enteras que se muestran en el cuadro 3.1.

tipo	Nro. Bytes	Nro. Bits	Rango de números permitido
byte	1 bytes	8 bits	$desde\ -(2^7+1)\ hasta\ +2^7$
short	2 bytes	16 bits	$desde\ -(2^{15}+1)\ hasta\ +2^{15}$
int	4 bytes	32 bits	$desde\ -(2^{31}+1)\ hasta\ +2^{31}$
long	8 bytes	64 bits	$desde\ -(2^{63}+1)\ hasta\ +2^{63}$

Cuadro 3.1: Tipos de variables enteras en Java

Para evaluar el tiempo de proceso de las operaciones de suma, multiplicación y división construimos un programa que repita muchas veces una

operación, midiendo el tiempo de ejecución. Esto permite determinar cuanto tiempo consumen los diferentes tipos de variables y escoger la más adecuada para mejorar el tiempo de proceso. El programa siguiente nos permite medir el tiempo que toma la operación de multiplicar realizando 10.000.000 de operaciones de multiplicación para una variable entera.

```
public class Tiempos {

    /**
     * Programa para medir el tiempo de proceso en
     * milisegundos de las operaciones elementales
     * Suma , Multiplicacion y division.
     * Se construye un ciclo, con una operacion elemental.
     * El ciclo debe ser lo suficientemente grande y se
     * imprime el tiempo
     *
     * @author Jorge Teran
     * @param args
     *            none
     */
    public static void main(String[] args) {
        int j = 0;
        int iteraciones = 10000000;
        long tiempoInicio, tiempoFin, tiempoTotal;
        tiempoInicio = System.currentTimeMillis();
        for (int i = 1; i < iteraciones; i++) {
            j = j + 1;
        }
        tiempoFin = System.currentTimeMillis();
        tiempoTotal = tiempoFin - tiempoInicio;
        System.out
                .println("Tiempo de proceso de la Suma "
                        + tiempoTotal);
        tiempoInicio = System.currentTimeMillis();

        for (int i = 1; i < iteraciones; i++) {
            j = j * j;
        }
```

Operacion	tipo	Tiempo en mseg para 10.000.000 repeticiones
suma	int	110
suma	long	190
multiplicación	int	140
multiplicación	long	230
división	int	771
división	long	2334

Cuadro 3.2: Comparación de los tiempos de ejecución de las operaciones básicas

```
        tiempoFin = System.currentTimeMillis();
        tiempoTotal = tiempoFin - tiempoInicio;
        System.out
            .println("Tiempo de proceso de la Multiplicacion "
                    + tiempoTotal);
        tiempoInicio = System.currentTimeMillis();
        for (int i = 1; i < iteraciones; i++) {
            j = j / i;
        }
        tiempoFin = System.currentTimeMillis();
        tiempoTotal = tiempoFin - tiempoInicio;
        System.out
                .println("Tiempo de proceso de la Division "
                        + tiempoTotal);
    }
}
```

Una vez procesado este programa construimos el cuadro 3.2 para comparar el tiempo de proceso para cada tipo de variable. Se puede observar que una operación de división demora mucho más que una operación de multiplicación. También se observa que las operaciones de tipo *long* demoran casi el doble de tiempo que las operaciones *enteras*.

Una alternativa en el sistema operativo Linux es el de utilizar la instrucción *time* para medir el tiempo de proceso. Esta instrucción de la línea de comandos nos da el tiempo de ejecución total del programa, en cambio, el método mostrado nos permite hallar el tiempo de ejecución de partes del

código.

3.3. Cálculo del máximo común divisor

Dados dos números enteros se denomina máximo común divisor al máximo número que es divisor de ambos números. Como ejemplo, tomemos los números 20 y 8. Los divisores del número 20 son: 20,10, 5,4 y 2. Los divisores del número 8 son el 8, 4 y 2. De aquí vemos que el máximo común divisor es el número 4. Para resolver adecuadamente este problema se construye un programa, inicialmente con la idea explicada, calculamos su $O(n)$ para posteriormente buscar un solución más eficiente.

Para realizar una primera aproximación a la solución del problema probaremos todos los números hallando el máximo número que divida a ambos.

```
Mcd(int a, int b) {
   int mcd() {
   int i;
   for (i=b; i>1;i--){
       if (((a%i==0) && (b%i==0))) {
          break;
       }
    }
   return (i);
  }
}
```

Análisis:

Se puede apreciar que primero $a > b$ cuando hallamos el primer divisor este ya es máximo porque comenzamos en b y vamos restando de uno en uno. En el peor caso cuando el máximo común divisor es 1 el algoritmo realiza el máximo número de operaciones. Para calcular el $O(n)$ requerimos resolver

$$\sum_{i=b}^{1} 1 = b$$

o sea $O(n)$. Para mejorar este algoritmo se hace necesario analizar algunas propiedades matemáticas.

3.3.1. Divisibilidad

En la teoría de números la notación $a|b$ se lee a divide b significa que $b = ka$ para algún $k > 0$. También se dice que b es múltiplo de a.

Si $a|b$ decimos que a es un divisor de b por ejemplo los divisores de 20 son: 1, 2,4,5 ,10 y 20.

Todo entero a, es divisible por el divisor trivial 1. Los divisores no triviales se denominan factores.

Cuando un número tiene como único divisor el número 1 y a si mismo se denomina número primo.

Teorema 1. *Para cualquier entero a y un número positivo n existen enteros k y r tal que $0 \leq r < n$ y $a = qn + r$*

El valor $q = \lfloor a/n \rfloor$ se denomina cociente de la división y r de denomina residuo o resto. El valor r también se expresa como $r = a \mod n$. De esto vemos que $n|a$ si y solo si $a \mod n = 0$

Si $d|b$ y $d|a$ entonces d es un común divisor de a y b. En este caso, se verifica

- $d|(a + b)$
- y en un caso más general $d|(xa + yb)$
- si $b|a$ y $a|b$ significa que $a = \pm b$
- si $b|a$ significa que $b \leq a$ o que $a = 0$

El máximo común divisor de dos números a, b ambos diferentes a cero, se denomina al divisor común más grande de a, b. Se lo denominamos como $mcd(a, b)$. Se define $mcd(0, 0) = 0$. Algunas propiedades son las siguientes:

- $mcd(a, b) = mcd(b, a)$
- $mcd(a, b) = mcd(-a, b)$
- $mcd(a, b) = mcd(|a|, |b|)$
- $mcd(a, ka) = a$
- $mcd(a, 0) = |a|$

Teorema 2. *Si a, b son números enteros, diferentes de cero el $mcd(a, b)$ es el elemento más pequeño del conjunto $\{ax + by\}$ donde $x, y \in Z$.*

Demostración. Sea $q = \lfloor a/s \rfloor$ y sea s el entero más pequeño de la combinación lineal de a y b. Entonces

$$\begin{aligned} a \mod s &= a - qs \\ &= a - q(ax + by) \\ &= a(1 - qx) + b(qy) \end{aligned}$$

Esto muestra que $a \mod s$ es una combinación lineal de a y b. El número más pequeño que puede dar $a \mod s$ es 0 dado que $a \mod s < s$. Podemos hacer un análisis similar y encontrar lo mismo para b. Esto indica que $s|a$ y que $s|b$ por lo que s es un divisor común de a y b.

De acá vemos que $mcd(a, b) \geq s$. Dado que el $mcd(a, b) > 0$ y que s divide a ambos a y b debe dividir también a una combinación lineal de a y b por lo que $mcd(a, b) \leq s$. Si $mcd(a, b) \geq s$ y $mcd(a, b) \leq$ entonces $mcd(a, b) = s$ □

De este teorema podemos deducir las siguientes propiedades:

- Dados los números enteros positivos a, b, n
  si $n|ab$ y $mcd(a, n) = 1$ entonces $b|n$.
- $mcd(na, nb) = n\ mcd(a, b)$ (propiedad distributiva).
- Si $d|a$ y $d|b$ entonces $mcd(a, b)|d$.

Teorema 3. *(Teorema de Euclides) Para cualesquiera dos enteros no negativos a, b*

$$mcd(a, b) = mcd(a, mcd(a \mod b))$$

Demostración. Se puede escribir a como $tb + r$ y reemplazando se tiene

$$mcd(a, b) = mcd(tb + r, b)$$

cualquier divisor común de b y a debe dividir $tb + r$. Cualquier divisor de tb también es divisor de b que implica que cualquier divisor de b debe dividir a r. □

Ahora vamos a aplicar el algoritmo denominado de Euclides para hallar el máximo común divisor de dos números. Primeramente escribiremos una función recursiva para hacer más simple el cálculo de su complejidad.

```
Mcd(a,b)
if (b==0)
   return (a)
   else Mcd(b, a%b)
```

Esta recursión la podemos escribir como

$$T(n) = \begin{cases} 1 & si\ n = 0, \\ T(\frac{n}{b}) + 1 & en\ otros\ casos. \end{cases}$$

Como ve toma la forma del teorema master

$$T(n) = aT(\frac{n}{b}) + n^k$$

donde b reduce su tamaño en $a \mod b$ en cada iteración por lo que podemos aplicar el teorema y se ve que $a = 1$, $b = a \mod b$, $k = 0$ donde $a = b^k$. Esto nos dice que el tiempo que toma algoritmo en su peor caso es $O(\log n)$.

Para ahorrar la memoria utilizada por la recursión se presenta una solución recursiva

```
public class Euclides {
    /**
     * Programa para hallar el maximo comun divisor
     * con el metodo de Euclides
     *
     * @author Jorge Teran
     */

    int a, b;

    Euclides(int a, int b) {
        if (a > b) {
            this.a = a;
            this.b = b;
        } else {
            this.b = b;
            this.a = a;
```

```
        }
    }

    int mcd() {
        int r = b;
        while (b > 0) {
            r = a % b;
            a = b;
            b = r;
        }
        return (a);
    }
}
```

Se mencionan algunas propiedades del máximo común divisor:

Propiedad asociativa. $mcd(a, b, c) = mcd(a, mcd(b, c))$

Propiedad distributiva. $mcd(na, nb) = n\ mcd(a, b)$

Propiedad idempotente. $mcd(a, b) = a)$

Propiedad conmutativa $mcd(a, b) = mcd(b, a)$

Propiedad de absorción $mcd(a, mcd(a, b)) = a$

3.3.2. Algoritmos extendido de Euclides

El algoritmo de Euclides Extendido muestra como calcular los valores x, y de la expresión $ax + by = mcd(a, b)$. Conocemos que

$$mcd(a, b) = mcd(b, r)$$

donde:

$$r = a - b\,\lfloor a/b \rfloor$$

suponiendo que conocemos x' y y' tal que

$$rx' + by' = mcd(b, r) = mcd(a, b)$$

reemplazando

$$(a - b\,\lfloor a/b \rfloor)x' + by' + = mcd(a, b)$$

reordenando

$$ax' + b(y' - y' \lfloor a/b \rfloor) = mcd(a, b)$$

Vemos que cuando $a = 0$ el valor de x debe ser 0 y $y = 1$

El siguiente programa realiza el cálculo de los valores x, y

```
/*Programa para el algoritmo
 * extendido de Euclides
 * @author Jorge Teran
 *
 */
void eMcd (int a, int b){
int x, yAnt, r, aIni, bIni, sr,q;
aIni  = a;
bIni  = b;
x = 1;
yAnt = 0;
while (b != 0)
{
    r  =  a % b;
    q=a/b;
    a  =  b;
    b  =  r;
    sr = x - yAnt*q;
    x = yAnt;
    yAnt = sr;
   }
   System.out.print("mcd= "+a);
   System.out.print(" x= "+x);
   System.out.println(" y= "+((a-x*aIni)/bIni));
}
```

Para una mayor comprensión se muestra un ejemplo de una ejecución. Hallemos $ax + by = mcd(a, b)$ con $a = 97$ y $b = 66$. El cuadro 3.3 ejemplifica la ejecución del programa.

Como se ve la ecuación $ax + by = mcd(a, b)$ se cumple con los últimos valores obteniendo $97 \cdot (-17) + 66 \cdot 25$

a	b	q=a/b	r	mcd	sr	x	y	97*x+66*y
97	66	1	31	66	1	0	1	66
66	31	2	4	31	-2	1	-1	31
31	4	7	3	4	15	-2	3	4
4	3	1	1	3	-17	15	-22	3
3	1	3	0	1	66	-17	25	1

Cuadro 3.3: Prueba de ejecución del algoritmo de extendido de Euclides

3.4. Mínimo común múltiplo

El mínimo común múltiplo (mcm) de dos números es el número más pequeño que es divisible por ambos números. Por ejemplo el mínimo común múltiplo entre 9 y 6 es 18. Para el cálculo del mínimo común múltiplo no existe un algoritmo similar al de Euclides que facilite hallar este número. Este número se expresa como sigue:

$$mcm(a, b) = min\{k, k > 0 \; y \; a|k \; y \; b|k\}$$

sin embargo es posible probar que

$$mcd(a, b)mcm(a, b) = ab$$

En el ejemplo podemos ver que el $mcd(9, 6)$ es 3 y que $3x18 = 54 = 9x6$

3.5. Aritmética modular

La aritmética modular es una de las aplicaciones más importantes de la teoría de números. Está representada por la función $\mod$. La función módulo representa el resto de la división. Por ejemplo $a \mod b$ significa que queremos hallar el resto de $\frac{a}{b}$ que representamos como

$$a \mod b = a - b \left\lfloor \frac{a}{b} \right\rfloor \tag{3.5.1}$$

La aritmética modular también define un sistema de numeración. Veamos por ejemplo la secuencia:
$0 \mod 3$, $1 \mod 3$, $2 \mod 3$, $3 \mod 3$, $4 \mod 3$....
evaluando tenemos $0, 2, 1, 0, 1$... que equivale a la numeración en base 3.

3.5.1. Propiedades

Presentamos algunas propiedades importantes de la aritmética modular.

Suma $(x + y) \mod m = (x \mod m + y \mod m) \mod m$. Por ejemplo $(8 + 7) \mod 3 = 15 \mod 3 = 0$ y por la propiedad de la suma $(8 \mod 3 + 7 \mod 3) \mod 3$

Resta La resta es solo la suma con valores negativos por los que $(x - y) \mod m = (x \mod m - y \mod m) \mod m$.

Multiplicación La multiplicación $xy \mod m = (x \mod m)(y \mod m) \mod m$. Esto se da debido a que la multiplación es simplemente una suma repetida.

División No existe el inverso de la multiplicación como la conocemos por ejemplo veamos $dx \mod m = dy \mod m$ se puede pensar que podemos simplificar d obteniendo $x \mod m = y \mod m$, que no se cumple en todos los casos. Veamos un ejemplo

$$6 \cdot 2 \mod 3 = 6 \cdot 1 \mod 3 = 0$$

si simplificamos el 6 tenemos $2 \mod 3 = 1 \neq 1 \mod 3 = 2$.

Solo es posible realizar estas simplificaciones cuando el $mcd(d, m) = 1$.

Existen muchas aplicaciones de la aritmética modular, por ejemplo en el calendario los días de la semana correponden a una aritmética módulo 7, las horas, minutos y segundos corresponden al módulo 60. Hallar el último dígito de un número decimal corresponde a una aritmética módulo 10.

El ejemplo que se presenta es extractado de [MAR03]. Hallar el último dígito del número 2^{100}. Para ésto no es necesario hallar el valor del exponente y luego obtener el último dígito se puede resolver como sigue:

$$\begin{aligned}
&2^3 \mod 10 = 8\\
&2^6 \mod 10 = (8 \cdot 8) \mod 10 = 4\\
&2^{12} \mod 10 = (4 \cdot 4) \mod 10 = 6\\
&2^{24} \mod 10 = (6 \cdot 6) \mod 10 = 6\\
&2^{48} \mod 10 = (6 \cdot 6) \mod 10 = 6\\
&2^{96} \mod 10 = (6 \cdot 6) \mod 10 = 6\\
&2^{100} \mod 10 = 2^9 6 \cdot 2^3 \cdot 2^1 \mod 10 = 6 \cdot 8 \cdot 2 \mod 10 = 6
\end{aligned}$$

3.5.2. Congruencias

Se dice que a es congruente b módulo m cuando $(a-b)/m = km$. Se escribe

$$a \equiv b \mod m$$

Esto equivale a que a, b son múltiplos de m Algunas propiedades de las congruencias son:

Propiedad reflexiva $a \equiv a$

Propiedad simétrica $a \equiv b \Rightarrow b \equiv a$

Propiedad transitiva $a \equiv b \equiv c \Rightarrow a \equiv c$

Suma $a \equiv b \; y \; c \equiv d \Rightarrow (a+c) \equiv (b+d) \mod m$

Multiplicación $a \equiv b$ y $c \equiv d \Rightarrow (ac) \equiv (bd) \mod m$ con b, c enteros

Potencias $a \equiv b \Rightarrow a^n \equiv b^n \mod m$

Si se quiere conocer si $2^n - 1$ es múltiplo de 3. Es decir que $(2^n - 1) \mod 3 = 0$. Podemos aplicar la propiedad de la potencia y escribir $2 \equiv -1 \mod 3$ de donde $2^n \equiv (-1)^n \mod 3$. Por lo tanto es múltiplo de 3 cuando n es par.

3.5.3. Inverso multiplicativo

Consideremos que $mcd(a, n) = 1$ entonces existen enteros b y c tales que $ba + cn = 1$. Esto se puede escribir como $ba \equiv 1 \mod n$. De aquí b se denomina inverso multiplicativo de a.

3.5.4. Solución de ecuaciones

La ecuación de congruencia de primer orden está definida como:

$$ax \equiv b(\mod m)$$

Estas ecuaciones tienen solución solo si $mcd(a, m)|b$ Para resolver este tipo de ecuación se sigue la siguiente estrategia.

- $ax - b = km$
- $ax = b + km$ y buscamos un k tal que $a|(b + km)$

- hallamos x

Ejemplos:

1. Resolver $6x \equiv 5(mod/3)$. como $mcd(6,3) = 2$ no es divisor de 5,no se tiene una solución.

2. Resolver $4x \equiv 10(mod/6)$

$$4x = 10 + k6$$

$$4x = 10 + 6 \; con \; k = 1$$

$$x = 4$$

si reemplazamos el valor de x encontrado tenemos:

$$4 \cdot 4 \equiv 6(\mod 6)$$

también pueden plantearse sistemas de ecuaciones de congruencias, por ejemplo, si recogemos una cantidad de cajas en las cuales vienen 3 equipos y tenemos 2 equipos sueltos. Por otro lado si recibimos la misma cantidad de equipos, en cajas en las cuales vienen 5 equipos y tenemos 1 equipo suelto. Se desea conocer cuantos equipos se recibieron. Esto se puede escribir como:

$$x \equiv 3(\mod 7) \tag{3.5.2}$$

$$5x \equiv 7(\mod 12) \tag{3.5.3}$$

si la solución de 3.5.2 es $x = 3 + k7$ para algún valor k podemos reemplazar x en la segunda ecuación obteniendo:

$$5(3 + k7) \equiv 7(\mod 12)$$

$$15 + 35k \equiv 7(\mod 12)$$

$$35k \equiv -8(\mod 12)$$

significa que $k = 8+12t$ para algún t reemplazando en 3.5.2 $x = 3+(8+12k)7$ que da $x = 59 + 84t$ de donde $x = 59$.

3.6. Números primos

La teoría de los números primos ha tenido un desarrollo muy intenso con las aplicaciones de criptografía. En esta sección le dedicamos un especial interés debido a las complejidades que lleva éste cuando se trabaja con números grandes.

Se define como primo el número entero positivo que es divisible solamente por 1 o por si mismo. Los primeros números primos son $2, 3, 5, 7, 11, 19...$ Como se ve el único número primo par es el número 2. El número 1 no es parte de los números primos, sin embargo, algunos autores amplian la definición para incluirlo.

Para muchas aplicaciones es necesario realizar un test de primalidad que significa verificar si el número es primo o no. Esto se puede realizar por divisiones sucesivas sin embargo no es un método muy adecuado cuando se trata de números grandes.

3.6.1. Generación de primos

Es posible probar la primalidad de un número n en un tiempo proporcional a $O(\sqrt{n})$ con el siguiente código de divisiones sucesivas

```
j = 2;
while (j * j <= n) {
  if ((i%j)==0)
      break; //no primo
      j++;
}
```

El problema que representa este método es la cantidad de divisiones que hay que realizar. Como ya vimos el tiempo que toma una división es mucho mayor al de la suma. Para convertir éstas divisiones en suma de números, la forma más fácil es a través del método denominado la criba de Eratóstenes.

La criba de Eratóstenes se contruye a partir de los múltiplos de los números como sigue

1. Se marcan los múltiplos de 2.

2. Luego los de 3, 5, 7, etc. sucesivamente.

Múltiplo de	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2			x		x		x		x		x		x	
3					x			x			x			x
5									x					x
Marcados			x		x		x	x	x		x		x	x

Cuadro 3.4: Ejemplo de una criba de Eratóstenes

3. Una vez completado el marcado los no marcados son los números primos.

Como se observa en el cuadro 3.4 se marcan en la primera pasada los múltiplos de 2, la segunda pasada los múltiplos de 3 y así sucesivamente. Al terminar el proceso los dígitos no marcados son los números primos. El siguiente programa muestra como contar cuantos números primos hay hasta 100.000.000.

```
import java.util.*;

/**
 * Programa contar los numeros primos hasta = 100,000,000
 * utilizando la Criba de Eratostenes
 *
 * @author Jorge Teran
 */
public class Criba {
    public static void main(String[] s) {
        int n = 100000000;
        long inicio = System.currentTimeMillis();
        BitSet a = new BitSet(n + 1);
        int contar = 0;
        int i = 2, j = 0;
        // construccion de la criba
        for (i = 2; i * i <= n; i = i + 1) {
            if (!a.get(i)) {
                for (j = i + i; j <= n; j = j + i) {
                    a.set(j);
                }
```

```
            }
        }
        // contar los numeros primos
        for (i = 2; i <= n; i++) {
            if (!a.get(i)) {
                contar++;
                // System.out.print(i+" ");
            }
        }
        long fin = System.currentTimeMillis();
        System.out.println(contar + " primos");
        System.out.println((fin - inicio)
                + " millisegundos");
    }
}
```

En la computadora demoró 78.013 milisegundos dando el resultado de 5.761.455 primos.

Los números primos que pueden calcularse a través de este método se consideran números pequeños. El número máximo depende de la memoria de la máquina, la implementación realizada y el lenguaje de programación. Vea que si almacena solo los números impares puede almacenar el doble de números en la criba.

La página http://primes.utm.edu/ proporciona un acceso importante a la teoría de números primos y publica la cantidad de primos por rango conocidos a la fecha, están detallados en el cuadro 3.5.

3.6.2. Factorización

La factorización de un número consiste en descomponer el mismo en sus divisores. Se demuestra que todos los factores son números primos. Analice que si uno de sus factores es un número compuesto también podría descomponerse en factores. Para una demostración más formal vean [THCR90].

La forma de expresar éstos factores es como sigue: $a_1^{n_1} a_2^{n_2} a_3^{n_3} ....$

Los factores primos de 168 son $2, 2, 2, 3, 7$ que se expresa como $2^3 \cdot 3 \cdot 7 = 168$. El siguiente código muestra el proceso de factorización de números. Hay que indicar que este proceso es aplicable a números pequeños.

Hasta	Cantidad de Primos
10	4
100	25
1.000	168
10.000	1.229
100.000	9.592
1.000.000	78.498
10.000.000	664.579
100.000.000	5.761.455
1.000.000.000	50.847.534
10.000.000.000	455.052.511
100.000.000.000	4.118.054.813
1.000.000.000.000	37.607.912.018
10.000.000.000.000	346.065.536.839
100.000.000.000.000	3.204.941.750.802
1.000.000.000.000.000	29.844.570.422.669
10.000.000.000.000.000	279.238.341.033.925
100.000.000.000.000.000	2.623.557.157.654.233
1.000.000.000.000.000.000	24.739.954.287.740.860
10.000.000.000.000.000.000	234.057.667.276.344.607
100.000.000.000.000.000.000	2.220.819.602.560.918.840
1.000.000.000.000.000.000.000	21.127.269.486.018.731.928
10.000.000.000.000.000.000.000	201.467.286.689.315.906.290

Cuadro 3.5: Cantidad de primos por rango conocidos a la fecha

```
/**
 * Calcula los factores primos de N Ejemplo java Factors 81
 *
 * @author Jorge Teran
**/

public class Factores {

    public static void main(String[] args) {
        long N = Long.parseLong(args[0]);
        //long N = 168;
        System.out
        .print("Los factores primos de:  "
                + N + " son: ");

        for (long i = 2; i <= N / i; i++) {

            while (N % i == 0) {
                System.out.print(i + " ");
                N = N / i;
            }
        }

        // Si el primer factor ocurre una sola vez
        if (N > 1)
            System.out.println(N);
        else
            System.out.println();
    }
}
```

3.6.3. Prueba de la primalidad

La prueba de primalidad es simple para números primos pequeños. Se puede hacer por divisiones sucecivas, aún cuando, es mejor utilizar una criba con números precalculados. Las dificultades en el cálculo de números primos radica cuando los números a tratar son grandes. Esto hace necesario buscar

otros métodos para determinar la primalidad de los mismos. Analizamos algunos métodos sin ahondar en las demostraciones que están ampliamente desarrolladas en la teoría de números y el área de criptografía.

3.6.4. Teorema de Fermat

Este teorema indica que todo número primo cumple la relación:

$$a^{n-1} \equiv 1 \mod n \; \forall a \leq n \tag{3.6.1}$$

por lo que parece suficiente realizar este cálculo para determinar la primalidad.

Desafortunadamente solo sirve para conocer si un número es compuesto dado que para algunos números primos no se cumple. Estos números se denominan números de Carmichael . Veamos por ejemplo el número compuesto 561:

$$2^{561-1} \mod 561 = 1$$

Algunos números que no cumplen este teorema son el 561, 1105, 1729. Estos números se denominan pseudo primos.

Para determinar si un número es compuesto podemos comenzar un algoritmo con $a = 2$ y el momento que no se cumpla el teorema podemos decir que el número es compuesto. En otros casos se indica que el número es probablemente primo.

La utilidad de este método se ve en el tratamiento de números grandes donde la facilidad de la exponenciación módulo es mucho más eficiente que la división.

3.6.5. Prueba de Miller - Rabin

.- Este algortimo tiene su nombre por los autores del mismo. Esta prueba provee un algoritmo probabílistico eficiente aprovechando algunas características de las congruencias.

Dado un entero n impar, hagamos $n = 2^r s + 1$ con s impar. Escogemos un número entero aleatoriamente y sea éste $1 \leq a \leq n$. Si $a^s \equiv 1 \mod (n)$ o $a^{2^j s} \equiv -1 \mod (n)$ para algún j, que esté en el rango $0 \leq j \leq r-1$, se dice que n pasa la prueba. Un número primo pasa la prueba para todo a.

Para utilizar este concepto se escoge un número aleatorio a. Si pasa la prueba, probamos con otro número aleatorio. Si no pasa la prueba decimos

que el número es compuesto. Se ha probado que la probabilidad de que, un número a pase la prueba, es de $\frac{1}{4}$ por lo que hay que realizar varias pruebas para tener más certeza.

El propósito de comentar este algoritmo es el hecho que la implementación de Java ya lo incluye en sus métodos para trabajar con números grandes por lo que, no desarrollaremos el algoritmo en extenso.

3.6.6. Otras pruebas de primalidad

En la (http://mathworld.wolfram.com/search/) enciclopedia de matemáticas Mathworld se pueden encontrar otras pruebas de primalidad tanto probabilísticas, como determinísticas de las que mencionamos: Curva elíptica, test de primalidad de Adleman-Pomerance-Rumely, AKS de Agragual, Lucas-Lehmer, Ward's y otros.

Muchos de éstos métodos no se han visto efectivos para la prueba de primalidad cuando se trata de números grandes.

3.7. Bibliotecas de Java

La aritmética de números grandes es esencial en varios campos tales como la criptografía. Las bibliotecas de Java incluyen una variedad de funciones para el tratamiento de éstos números.

- Constructores

  BigInteger(String val) Permite construir un número grande a partir de una cadena

  BigInteger(int largo, int certeza, Random r) Permite construir un número probablemente primo de la longitud de bits especificada, la certeza especificada y random es un generador de números aleatorios.

- Métodos

  add(BigInteger val) Devuelve $this + val$

  compareTo(BigInteger val) Compara $this$ con val y devuelve $-1, 0, 1$ para los casos que sean $<, =, >$ respectivamente

  intValue() Devuelve el valor entero de $this$

gcd(BigInteger val) Halla el máximo común divisor entre *this* y *val*

isProbablePrime(int certeza) .- Prueba si el número es probablemente primo con el grado de certeza especificado. Devuelve falso si el número es compuesto.

mod(BigInteger m) Devuelve el valor $this \mod m$

modInverse(BigInteger m) Retorna el valor $this^{-1} \mod m$

modPow(BigInteger exponente, BigInteger m) Devuelve $this^{exponente} \mod m$

multiply(BigInteger val) Devuelve $this * val$

pow(int exponente) Devuelve $this^{exponente}$

probablePrime(int longitud, Random rnd) Devuelve un número probablemente primo. El grado de certeza es 100. Y *rnd* es un generador de números aleatorios.

remainder(BigInteger val) Devuelve $this \% val$

subtract(BigInteger val) Devuelve $this - val$

toString(int radix) Devuelve una cadena en la base especificada.

3.7.1. Ejemplo

Hallar un número primo aleatorio de 512 Bits. La función probablePrime utiliza para ésto las pruebas de Miller-Rabin y Lucas-Lehmer.

```
/**
  *Programa para hallar un numero
  * primo de 512 bits
  * @author Jorge Teran
  *
 */

import java.math.BigInteger;
import java.util.Random;

   public class PrimoGrande {
    public static void main(String args[]) {
```

```
      Random rnd = new Random(0);
      // generar un probable primo de 512 bits
      // con una probabilidad de que sea compuesto
      // de 1 en 2 elevado a 100.
      BigInteger prime =
         BigInteger.probablePrime(512, rnd);

      System.out.println(prime);
   }
}
```

Un ejemplo de ejecución del programa es:

```
11768683740856488545342184179370880934722814668913767795511
29768394354858651998578795085800129797731017311099676317309
35911488339878356533264880532790710 57
```

Veamos otro ejemplo. Supongamos que queremos desarrollar el algoritmo de encriptación de llave pública RSA . El algoritmo consiste en hallar dos números d, e que cumplan la propiedades siguientes: $c = m^e \ mod \ n$ y $m = c^d \ mod \ n$. Para resolver esto procedemos como sigue:

1. Escoger dos números primos p y q con $p \neq q$

   ```
   BigInteger p = new BigInteger(longitud/2, 100, r);
   BigInteger q = new BigInteger(longitud/2, 100, r);
   ```

2. Calcular $n = pq$

   ```
   n = p.multiply(q);
   ```

3. Escoger e que sea relativamete primo a (p-1)(q-1)

   ```
   BigInteger m = (p.subtract(BigInteger.ONE))
         .multiply(q.subtract(BigInteger.ONE));
   e = new BigInteger("3");
   while(m.gcd(e).intValue() > 1)
      e = e.add(new BigInteger("2"));
   ```

4. Calcular el multiplicativo inverso de e

```
    d = e.modInverse(m);
```

5. Publicar la clave pública como el par $p = (e, n)$
6. Publicar la clave privada como el par $s = (d, n)$

Veamos el código completo. Aquí hay que hacer notar que en Java existen dos clases para hallar números aleatoreos, *Random* y *SecureRandom*. La difrencia entre ambas está en que la clase *SecureRandom* hace la semilla inicial de generación de números menos predecible. Esto significa que de una ejecución a otra la secuencia aleatoria no va a ser la misma.

```
import java.math.BigInteger;

public class Principal {

  /**
   * Programa para utilizar la Clase RSA
   * @author Jorge Teran
 * @param args ninguno
   */
  public static void main(String[] args) {
    Rsa a = new Rsa (100); //Hallar d y e de 100 Bits
    BigInteger mensaje=new BigInteger("64656667686970");
    BigInteger cifrado= a.cifrado(mensaje);
    BigInteger limpio= a.decifrado(cifrado);
    System.out.println(mensaje);
    System.out.println(cifrado);
    System.out.println(limpio);
  }
}

import java.math.BigInteger;
import java.security.SecureRandom;

class Rsa
{
    private BigInteger n, d, e;
    public Rsa(int longitud){
```

```
      SecureRandom r = new SecureRandom();
      BigInteger p = new BigInteger(longitud/2, 100, r);
      BigInteger q = new BigInteger(longitud/2, 100, r);
      n = p.multiply(q);
      BigInteger m = (p.subtract(BigInteger.ONE))
                  .multiply(q.subtract(BigInteger.ONE));
       e = new BigInteger("3");
       while(m.gcd(e).intValue() > 1)
            e = e.add(new BigInteger("2"));
       d = e.modInverse(m);
  }
  public BigInteger cifrar(BigInteger mensaje)
  {
    return mensaje.modPow(e, n);
  }
  public BigInteger decifrar(BigInteger mensaje)
  {
     return mensaje.modPow(d, n);
  }
}
```

3.8. Ejemplos de aplicación

Del juez virtual de la Universidad de Valladolid
http://online-judge.uva.es/. Problema 136 - Números Feos que resolveremos para mostrar dos estrategias a resolver el problema. El problema dice:

Los números feos son números cuyos únicos factores primos son 2, 3, o 5.

La secuencia:
$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ...$
Muestra los primeros 11 números feos. Por convención se incluye el 1.

Esto equivale a encontrar los números que se pueden formar por $2^a 3^b 5^c$. Escriba un programa que encuentre e imprima el número 1500.

Entrada y salida.- No existe entrada en este problema. La salida debe consistir de una línea reemplazando *número* con el número calculado.

Ejemplo de salida The 1500'th ugly number is *número*.

A fin de resolver el problema una primera aproximación puede ser el de ir probando uno a uno todos los números y hallar sus factores primos hasta

hallar el número 1500. Con esta estrategia escribimos el siguiente programa:

```
/**
 * Solucion lenta al problema de numero feos
 * por factorizacion
 *
 * @author Jorge Teran
 **/
import java.io.*;

public class SolFact {
    public static void main(String[] args)
            throws IOException {
        int feo = 0;
        long nlong = 0;
        while (feo < 1500) {
            nlong++;
            if (Factores(nlong)) {
                feo++;
            }
        }
        System.out
                .println("The 1500'th ugly number is "
                        + feo + ".");
    }

    // Calcula los factores primos de N
    static boolean Factores(long N) {
        for (long i = 2; i <= 5; i++) {
            while (N % i == 0) {
                N = N / i;
            }
        }
        if (N > 1)
            return false;
        else
            return true;
    }
```

```
}
```

Si analizamos que el resultado es 859963392 significa que hemos tenido que procesar $859,963,392$ veces la rutina de hallar los factores. Por este motivo es más conveniente adoptar una estrategia diferente. Construyamos los números feos.

Para construir los números acudimos a la definición: son los números que cumplen $2^a 3^b 5^c$. Podemos construir estos y almacenarlos en un vector. Luego ordenamos el vector e imprimimos el resultado. Esto nos lleva al programa siguiente:

```
/**
 * Solucion al problema numeros feos
 *
 * @author Jorge Teran
 **/
import java.util.*;

public class NumerosFeos {

    public static void main(String[] args) {
        int cinco = 13, tres = 19, dos = 30;
        int max = dos * tres * cinco;
        long[] v = new long[max];
        int pos = 0;
        for (int i = 0; i < dos; i++)
            for (int j = 0; j < tres; j++)
                for (int k = 0; k < cinco; k++)
                    v[pos++] = (long) (Math.pow(
                            2, i)
                            * Math.pow(3, j) * Math
                            .pow(5, k));
        Arrays.sort(v);
        System.out
                .println("The 1500'th ugly number is "
                        + v[1499] + ".");
    }
}
```

Analizando este programa vemos que solo tuvimos que generar $13 * 19 * 30 = 7410$ operaciones de potencia ordenar e imprimir obteniendo un ahorro significativo en el tiempo de proceso.

Analicemos otro ejercicio, esta vez veamos el ejercicio 11408 (deprimes) del Juez de Valladolid.

Un número se denomina *deprimo* si la suma de sus factores primos es un número primo. Dados los números a, b cuente cuantos números *deprimo* existen entre a, b donde $2 < a < b \leq 5000000$.

Entrada y salida.- La entrada consiste en los números a, b separados por un espacio. Termina cuando $a = 0$.

La salida es un solo número que indica cuantos números *deprimo* existen entre a, b.

Ejemplo de salida

```
2 5
10 21
100 120
0
```

Ejemplo de salida

```
4
9
9
```

Veamos un ejemplo:

Número	Factores Primos
2	2
3	3
4	2
5	5

La cantidad de números cuya suma de sus factores en un número primo es 4. Si tomamos el número 14 tiene dos factores primos el 2 y el 7 y la suma da 9 que no es un número primo.

Una primera aproximación al problema sería: definir una criba para determinar la primalidad, crear una función para descomponer un número en sus factores primos. Luego contar para obtener la solución. Esto nos llevaría al programa siguiente.

```
/**
 * Solucion lenta al problema Deprimos
 *
 * @author Jorge Teran
 **/
import java.util.Scanner;

public class DePrimo {

    public static int n = 5000000;
    public static int[] p = new int[n + 1];

    public static void main(String[] args) {
        int contar = 0;
        int i = 2, j = 0, a, b;
        // construccion de la criba
        for (i = 2; i * i <= n; i = i + 1)
            // funciona en criba
            if (p[i] == 0) {
                for (j = i + i; j <= n; j = j + i) {
                    p[j] = 1;
                }
            }
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {
            contar = 0;
            a = in.nextInt();
            b = in.nextInt();
            for (i = a; i <= b; i++) {
                if (p[i] != 0) {
                    j = Factores(i);
                    if (p[j] == 0) {
                        contar++;

                    }
                } else
                    contar++;
            }
```

```
            System.out.println(contar);
        }
    }

    static int Factores(int N) {
        int s = 0;
        for (int i = 2; i <= N / i; i++) {
            if (p[i] == 0) {
                if (N % i == 0) {
                    s = s + i;
                }
                while (N % i == 0) {
                    N = N / i;
                }
            }
        }
        if (N != 1) {
            s = s + N;
        }
        return s;
    }
}
```

En este programa en lugar de *bitset* simplemente hemos utilizado un vector de números enteros para construir la criba. Cuando ejecutamos el programa vemos que produce los resultados correctos, pero cuando ponemos los límites entre 2 y 5000000 el tiempo excede en mucho los tres segundos descritos en el enunciado como límite de tiempo para la ejecución.

Para probar que es posible mejorar el tiempo de ejecución, recurrimos al sitio para pruebas http://uvatoolkit.com/problemssolve.php y probamos el problema con el rango máximo. La imagen 3.8 siguiente nos muestra lo que deberíamos ver:

Para mejorar el código y obtener un tiempo de ejecución más eficiente recurrimos a analizar la criba. En ella hemos registrado cuales son números primos y cuales compuestos. La idea sería almacenar la suma de los factores. Si analizamos el algoritmo de construcción vemos que se van primero marcando los múltiplos de 2 luego los de 3 y así sucesivamente. En lugar de marcar los múltiplos los sumamos. Con esto ya no será necesario descompo-

Figura 3.1: Prueba el uvatoolkit.com

ner los mismos en sus factores primos. El programa que obtendríamos es el siguiente:

```
/**
 * Solucion final al problema Deprimos
 *
 * @author Jorge Teran
 **/
import java.util.Scanner;

public class DePrimoFinal {

    public static int n = 5000000;
    public static int[] p = new int[n + 1];

    public static void main(String[] args) {
        int contar = 0;
        int i = 2, j = 0, a, b;
        // construccion de la criba
        for (i = 2; i <= n; i++)
            if (p[i] == 0) {
                for (j = i + i; j <= n; j = j + i) {
                    p[j] += i;
                }
            }
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {
```

```
            contar = 0;
            a = in.nextInt();
            b = in.nextInt();
            for (i = a; i <= b; i++) {
                if (p[i] != 0) {
                    j = p[i];
                    if (p[j] == 0) {
                        contar++;

                    }
                } else
                    contar++;
            }
            System.out.println(contar);
        }
    }
}
```

Como ve una pequeña variación a la criba puede proporcionar resultados adicionales muy interesantes.

3.8.1. Ejercicios

Resolver los siguientes ejercicios del Juez de Valladolid son relacionados a teoría de números.

106, 128, 160, 256, 294, 332, 343, 356, 374, 382, 386, 406, 412, 443, 485, 516, 524, 530, 543, 547, 568, 580, 583, 636, 686 701, 713, 756, 847, 884, 903, 911, 913, 914, 967, 974, 995, 10006, 10015, 10025, 10036, 10042, 10049, 10057, 10061, 10077, 10083, ,10090, 10093, 10104, 10105, 10109, 10110, 10127, 10139, 10140, 10161, 10162, 10168 10170, 10174, 10176, 10179, 10193, 10200, 10209, 10212, 10235, 10299, 10311, 10323, 10325, 10329, 10365, 10367, 10368, 10387, 10392, 10394, 10407, 10408, 10450, 10489, 10490, 10523, 10527, 10533, 10539, 10551, 10606, 10616, 10622, 10627, 10633, 10650, 10673, 10680, 10694, 10699, 10705, 10706, 10710, 10717, 10719, 10738, 10780, 10787, 10789, 10791, 10799, 10814, 10820, 10830, 10831, 10852, 10862, 10892, 10922, 10924, 10925, 10929, 10931, 10948, 10951, 10990, 11005, 11064, 11105, 11121, 11155, 11180, 11185, 11191, 11226, 11237, 11245, 11254, 11256, 11273, 11287, 11312, 11327, 11344, 11347, 11350, 11353, 11361, 11371, 11388, 11392, 11395, 11408, 11415, 11417,

11424, 11426, 11428, 11430, 11436, 11440, 11444, 11448, 11461, 11466, 11489, 11610, 11628, 11633, 11728

Capítulo 4

Codificación con miras a la prueba

4.1. Introducción

La prueba de programas de software ha sido siempre un campo muy árido y no se ha tomado en cuenta en el momento del desarrollo del código. Para desarrollar programas más fiables se hace necesario planificar la verificación del código al momento de su construcción. Durante muchos años se han prometido programas que podrían verificar a otros. Desde la década de los 60 seguimos esperando. Aún no existe esta posibilidad.

Hoy en día se hace mucho más importante la verificación de los programas. Existen programas en diferentes industrias que, en caso de fallas pueden causar inclusive la muerte de personas y otros que no permiten pruebas experimentales.

Como ejemplo, podemos mencionar los programas que controlan el vuelo de los aviones, servomecanismos asistidos por computadora e inclusive muchos de los electrodomésticos que existen en su hogar.

Tradicionalmente la prueba del software ha sido presentada como un detalle de pruebas denominadas de caja negra y caja blanca. Las pruebas de caja negra son aquellas que se enfocan en el exterior del módulo sin importar el código. Las pruebas de caja blanca son más amplias y tratan de verificar el código probando partes indicando la cobertura del código probado. Estos planes de prueba han dado como resultado que durante la explotación del software siguen apareciendo errores no detectados en la prueba.

Hoy en día se tiene un conocimiento más profundo de la programación y se pueden realizar muchas más verificaciones que, solo las de la caja negra que pueden indicarnos bien o mal.

El control de calidad y la verificación de programas nos permite encontrar errores de los programas. La codificación es una de las habilidades requeridas para esto.

En el presente capítulo presentaremos una serie de técnicas que ayudan a la prueba y mantenimiento del código. No se trata de ninguna manera de entrar en el campo de la verificación formal de programas, sin embargo, espero que sirva de una buena introducción. Se presenta la especificación formal, las aserciones y la programación por contrato.

4.2. Algunos errores de programación

Cuando revisaba las revistas Comunicatión of the ACM [KBO+05] encontré una serie de preguntas sobre errores existentes en diferentes códigos. Estos errores normalmente no son vistos por los programadores proveen una puerta por donde se producen fallos mal intencionados durante la explotación del programa. Por este motivo quiero presentar este tema.

El desborde de la memoria se produce cuando se accede fuera de las dimensiones de la variable definida. Por ejemplo salirse del tamaño de un vector incrementando el tamaño del índice fuera de los límites del mismo.

Veamos un ejemplo:

```
void copia(char[] b) {
   char[] datos = new char(100);
   for (int i=1;i< b.length; i++)
      datos[i]=b[i]
      ..
      ..
      ..
 }
 return
```

En el segmento de código expuesto se puede apreciar que se copia el vector *b* a *datos* pudiendo causarse un desborde en *datos* si es que *b* es de mayor tamaño que *datos*.

Normalmente, durante la ejecución, se obtendrá un error de *índice fuera de rango*. Cuando colocamos un programa en producción generalmente se recompilan los códigos con argumentos de optimización y el control de índices fuera de rango queda eliminado. En estos casos al no producirse este error se obtienen mensajes tales como *error de segmentación*, *error de bus*.

Estos errores se producen al reescribir parte del código con datos, al salirse del rango del vector *datos*. Como, se va ha reescribir parte del código, un intruso puede enviar una serie de códigos que quiere ejecutar.

Veamos un ejemplo escrito en lenguaje C.

```
//programa para desplegar el primer
// argumento de la línea de comando
// desp.c
// Autor Jorge Teran
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
   int main(int argc, char *argv[])
   {     char copia[2];
         strcpy(copia,argv[1]);
         printf("%s\n", copia);
         return 0;
   }
```

Como se ve este programa despliega en pantalla el argumento introducido en la línea de comando, y la ejecución del mismo se ve así:

```
>desp 12
12
```

Que pasa si en la invocación al mismo colocamos más de dos dígitos?

```
>desp 12345
12345
```

Cuando llegamos a 6 dígitos obtenemos una lista interminable de

```
>desp 123456
123456
123456
123456
....
```

Claramente hemos invadido el área de código y eliminado parte de la instrucción *return* de la rutina printf. Si introducimos más valores se obtienen otros errores. Como se dijo, ésto puede ser un mecanismo para introducir códigos a un programa y hacer que realice algo para lo que no ha sido diseñado.

Es posible que se cambie el código para realizar un chequeo de *índice fuera de rango* durante la copia pero, consume más tiempo de proceso.

Lo que queremos expresar con este ejemplo es que aún cuando un programa haya sido probado y se supone que funciona correctamente puede presentar fallas. Tal es el caso de la rutina *strcpy* mostrada.

Con respecto al lenguaje de programación podemos preguntarnos cual lenguaje ofrece mejores condiciones en el tiempo de proceso durante la ejecución del programa. La respuesta es bastante obvia. Cuanto más controles se tengan se demorará más tiempo de proceso. Y en caso contrario se tienen algunas inseguridades a cambio de una mejora en la ejecución.

En el caso del lenguaje Java al ejecutarse en un entorno cerrado provee control sobre la ejecución del código que es inexistente en el lenguaje C. Esto hace que este tipo de problemas no se presenten en algunos lenguajes como el Java.

Existen algunas soluciones que se pueden adoptar para resolver este tipo de problemas por ejemplo, introducir un control de subíndices durante la copia, proteger la memoria donde se encuentra la dirección de retorno a solo lectura. Estas acciones hacen el tiempo de proceso más lento.

La programación Java no está exenta de fallas que se producen en la ejecución del código, por situaciones no previstas. Para ejemplificar supongamos la clase *Nombre* que tiene un método para devolver el nombre.

```
class Nombre {
  String nombre;
  Nombre(String n) {
  nombre=n;
  }
```

```
  String devuelveNombre() {
    return (nombre);
  }
}
```

Suponga que quiere utilizar la clase *Nombre* como sigue:

```
Nombre persona = new Nombre(nombrePersona);
int largo=persona.devuelveNombre().trim().length();
```

Analizando vemos que se trata de calcular la longitud del nombre.

¿Qué pasa cuando *nombrePersona* es un valor nulo?

Claramente obtendremos un error de ejecución indicando la línea donde se produjo el error. El mensaje del sistema no dice cual es la causa del error.

Para solucionar este problema en medio de la ejecución de un sistema se tiene que corregir el programa para tomar en cuenta este posible error modificando el código como sigue:

```
Nombre persona = new Nombre(nombrePersona);
int largo=0;
String dato = persona.devuelveNombre();

if  (dato == null)
{
    System.err.println("Error grave:
        Falta la propiedad 'nombre' ");
}
else
{
  largo=dato.trim().length();
}
```

Ahora estamos seguros que el programa funciona y que, cuando da el error identifica claramente el problema. Bien supongamos que, ahora que está seguro que el programa no falla se pone el programa en ejecución en un sistema distribuído, puede ocurrir que no se encuentre la clase *persona* y el sistema falle. Para esto hay que volver a corregir el código adicionando un control:

```
Nombre persona = new Nombre(nombrePersona);

if  (persona == null)
{
    System.err.println(
       "Error grave: clase Persona no disponible");
}

int largo=0;

String dato = persona.devuelveNombre();

if  (dato == null)
{
    System.err.println(
    "Error grave: Falta la propiedad 'nombre' ");
}
else
{
 largo=dato.trim().length();
}
```

Estas consideraciones hacen necesario especificar el código de una forma que, se pueda determinar una serie de problemas con mucha anticipación.

4.3. Especificación de programas

La especificación de programas tienen su origen en Hoare [Hoa83] quien desarrolló las bases para la especificación formal de programas. También puede ver un excelente trabajo que describe esta axiomática en [CK79].

Cuando se especifica un programa hay que considerar que no solo puede tener errores, también puede ser que esté equivocada la especificación del mismo. Por esta razón también deben verificarse las especificaciones de los mismos. Aunque no es parte del texto un tratamiento riguroso de la verificación y prueba de programas, es necesario introducir este tema que se estudia en los cursos de métodos formales.

Precondición Son lo valores iniciales que toman las variables del programa, o también los pre requisitos necesarios para su correcto funcionamiento. Por ejemplo puede ser que una variable no sea nula, que los datos esten ordenados, la existencia de una clase, etc.

Post condición Son los valores finales que toman las variables del programa

Todo programa consiste de tres partes:

1. Inicialización
2. Preservación de propiedades
3. Terminación

La preservación de propiedades se denomina invariante y es lo que nos permite verificar el código.

Llamemos a $\{P\}$ precondición, a $\{Q\}$ post condición y C un programa. La notación

$$\{P\}C\{Q\}$$

significa que si partimos de un estado P después de que el programa termine llegaremos al estado Q. Esta notación se debe a Hoare [Hoa83].
Ejemplo:

$$\{X = 1\}X = X + 1\{X = 2\}$$

Este pequeño código especifica que si X es inicialmente 1 una vez corrido el programa X toma el valor de 2 que es claramente verdadero.

Normalmente en esta notación las precondiciones y las variables se escriben en letras mayúsculas. Las letras minúsculas se utilizan para especificar valores. Por ejemplo para indicar que la variable X toma una valor arbitrario x se escribe $X = x$.

Una expresión del tipo $\{P\}C\{Q\}$ es una especificación de correctitud parcial porque para verificar que es correcta, no es necesario que el programa C termine.

Una especificación más fuerte es la especificación de correctitud total en la que se pide la prueba de que el programa termine. La notación que se utiliza para esto es:

$$[P]C[Q]$$

la relación que existe entre ambas definiciones es:

$$correctitud\ total = terminacion + correctitud\ parcial$$

Por ejemplo si queremos especificar la correctitud total de que los valores de X, Y se intercambian podemos escribir:

$$[X = x \wedge Y = y]\ R = X;\ X = Y;\ Y = R\ [X = y \wedge Y = x]$$

No todas las variables deben especificarse en una precondición, veamos como ejemplo una división por restas sucesivas:

$$\begin{aligned}
&\{Y = y \wedge X = y\}\\
&R = X;\\
&Q = 0;\\
&WHILE\ Y \leq R\\
&\ R = R - Y; Q = Q + 1\\
&\{R < y \wedge X = R + (YxQ)\}
\end{aligned}$$

Este ejemplo es menos intuitivo para la verificación pero nos indica claramente que el resultado depende de los valores de X y Y. El valor de Q y R están especificados solo en la post condición.

Como no se especificaron en la precondición no se puede decir nada de ellos antes de la ejecución del código.

4.3.1. ¿Por qué especificar?

La primera pregunta que uno se hace es *por qué especificar?*. Uno puede especificar para tener mayor documentación del sistema, desea una descripción más abstracta o porque desea realizar un análisis del mismo.

Lo que uno debe preguntarse es porque especificar *formalmente* y las respuestas que se pueden conseguir de desarrolladores profesionales son:

1. Mostrar que una propiedad se mantiene globalmente en un programa
   - Se quiere caracterizar la condición de correctitud.
   - Se quiere mostrar que la propiedad es realmente una invariante.
   - Se quiere mostrar que mi sistema cumple algún criterio de alto nivel en el diseño.

2. Manejo de errores
    - Se quiere especificar que ocurre cuando se produce un error.
    - Se quiere especificar que las cosas correctas ocurren cuando se produce un error.
    - Se quiere estar seguro que no exista este error.
3. Completitud
    - Se quiere estar seguro que se han cubierto todos los casos, incluyendo los casos de error.
    - Se quiere saber si el programa que he diseñado es computacionalmente completo.
4. Especificar interfases
    - Se quiere definir una jerarquía de clases.
    - Se quiere una descripción más formal de la interfase del usuario.
5. Manejar la complejidad
    - El diseño está muy complicado para manejar todo en la cabeza y necesitamos una forma de pensar en pedazos más pequeños.
    - Cumplir los requisitos legales de algunos países.
6. Cambiar el control
    - Cada vez que cambio un pedazo de código quisiera conocer que otras partes son afectadas sin mirar todos los módulos y sin mirar el código fuente.

4.3.2. ¿Qué especificar?

Los métodos formales no están desarrollados de tal forma que un sistema entero y muy grande pueda ser especificado completamente. Solo se pueden especificar algunos aspectos tales como funcionalidad y comportamiento de un sistema de tiempo real.

Al escribir una especificación uno debe decidir si la descripción es requerida y permitida. ¿Puede o debe? Una vez que uno tiene claro lo que hay que especificar uno puede determinar que puede formalizarse.

Condiciones de correctitud

Esto implica la noción de una condición global de correctitud que el sistema debe mantener.

Invariantes

La forma de caracterizar algunas condiciones que no cambian de un estado a otro se denominan *invariantes*. Las invariantes son una buena forma de caracterizar las propiedades deseadas de un sistema. Formalizar las transiciones le permite probar que se mantienen durante la ejecución del programa.

Comportamiento observable

Se denomina cuando se especifica el comportamiento del sistema cuando interactúa con su ambiente.

Propiedades de entidades

Las propiedades más importantes de las entidades se expresan comunmente por el tipo. La correspondencia entre este tipo de propiedades se verifica por el compilador. Sin embargo, para entidades estructuradas podemos ver otras propiedades de las que, podemos anotar por ejemplo:

- Orden. ¿Los elementos están ordenados?
- Duplicados. ¿Se Permiten?
- Rangos. ¿Pueden acotarse los elementos de alguna manera?
- Acceso asociativo. ¿Los elementos se recuperan por índices o llaves?
- Forma ¿El objeto tiene una estructura lineal, jerárquica, arbitraria, gráfica, etc?

4.3.3. ¿Cómo especificar?

Dado que entiende lo que desea especificar y lo que desea obtener puede empezar a especificar. La técnica principal es la *descomposición* y *abstracción*. Para ésto damos algunas ideas:

- Abstraer. No piense como programador.
- Piense en función de la definición, no de la funcionalidad.
- Trate de construir teorías, no solo modelos.
- No piense en forma computacional-

4.3.4. Invariantes

Definamos con más claridad una invariante. Una invariante es una propiedad que es verdadera y no cambia durante la ejecución del programa. Esta propiedad engloba a la mayor parte de los elementos o variables del programa. Cada ciclo tiene una propiedad invariante. Las invariantes explican las estructuras de datos y algoritmos.

Las propiedades de las invariantes deben mantenerse cuando se modifica el programa. La invariante obtenida se utiliza en un proceso estático para verificar la correctitud del programa.

Las invariantes son útilies en todos los aspectos de la programación: diseño, codificación, prueba, mantenimiento y optimización. Presentamos una lista de usos a fin de motivar a la utilización de las mismas por los programadores.

- Escribir mejores programas.- Muchos autores coinciden que se pueden construir mejores programas cuando se utilizan especificaciones en el diseño de los programas. Las invariantes formalizan en forma precisa el contrato de un trozo de código. El uso de invariantes informalmente también ayuda a los programadores.
- Documentar el programa.- Las invariantes caracterizan ciertos aspectos de la ejecución del programa y proveen una documentación valiosa sobre el algoritmo, estructura de datos y los conocimientos a un nivel necesario para modificar el programa.
- Verificar las suposiciones.- Teniendo las invariantes las podemos utilizar en la verificación del programa observando que no cambian a medida que se procesa.
- Evitar errores.- Las invariantes pueden prevenir que el programa realice cambios que accidentalmente cambien las suposiciones bajo las cuales su funcionamiento es correcto.

- Formar un espectro.- El espectro de un programa son propiedades medibles en el programa o su ejecución. Algunas de éstas son el número de líneas ejecutadas por el programa, el tamaño de la salida o propiedades estáticas como la complejidad. La diferencia entre varios aspectos nos muestran cambios en los datos o el programa.

- Ubicar condiciones inusuales.- Condiciones inusuales o excepcionales deben llamar la atención a los programadores porque pueden ser causa de posibles errores.

- Crear datos de prueba.- Las invariantes pueden ayudar en la generación de datos de prueba de dos maneras. Pueden infringir intencionalmente las invariantes ampliando los datos de prueba. La otra es de mantener las invariantes para caracterizar el uso correcto del programa.

- Pruebas.- Prueba de teoremas, análisis de flujo de datos, chequeo del modelo y otros mecanismos automatizados o semi automatizados pueden verificar la correctitud del programa con respecto a su especificación.

Ejemplos

1. Mostrar que la invariante es $PX^N = x^n \wedge x \neq 0$

```
P:=1;
IF not (x=0)
    while not (N=0)
        IF IMPAR(N) P=PxX
        N=N /2
        X=X x X
ELSE
    P=0
```

   aquí la precondición es $\{X = x \wedge N = n\}$ y la post condición $\{P = x^n\}$. Veamos varios detalles del programa:

   - Tiene tres variables P, N, X.
   - Los valores iniciales son un valor x, n cualesquiera, precondición.
   - La post condición indica que el resultado es x^n.

Iteración	N	P	X	PX^N
1	9	1	2	512
-	9	2	2	–
-	4	2	4	512
2	2	2	4	–
-	2	2	16	512
3	1	2	16	–
-	1	2	256	512
4	1	512	256	512
-	0	512	256	–
-	0	512	65536	–

Cuadro 4.1: Valores que toman la variables durante la ejecución

Para mostrar que PX^n es una invariante tomamos dos valores X, N. Supongamos que $X = 2$ y $N = 9$. Con estos valores contruimos el cuadro 4.1 que muestra los valores que toman las variables durante la ejecución.

Como ve hemos anotado todos los valores que toman las variables involucradas y anotado los valores que toman al final de cada iteración calculando la supuesta invariante . Comprobamos que es correcta porque sus valores no cambian.

2. Las invariantes pueden tomar diferentes formas, consideremos por ejemplo el siguiente programa
precondición $\{M \geq 1 \wedge N = n\}$

```
X=0;
for (N=1; N<M; M++)
    X=X+M
```

post condición $\{X = (M(M-1))/2\}$

En este ejemplo vemos que efectivamente X en cualquier momento representa el resultado de la sumatoria hasta el número de iteración en que se encuentra. Esto indica que la invariante es:

$$X = \sum_{i=0}^{N-1} i$$

3. Indicar cuales son las precondiciones y post condiciones de un ciclo for.

```
for (I=1; I<=N; N++{
....
....
}
```

   Aquí vemos que I debe ser una variable definida, al principio del ciclo toma el valor de 1 y al terminar el mismo toma en valor de $N+1$. No se ha caracterizado una invariante dado que no se ha escrito un programa al interior del ciclo.

4. Considere el algoritmo de Euclides para hallar el máximo común divisor y caracterice cuál es la invariante del problema.

```
Mcd(a,b)
if (b==0)
   return (a)
   else Mcd(b, a%b)
```

   Recordando el teorema de Euclides, dados dos números a, b se cumple

$$Mcd(a, b) = Mcd(b, a\,\%b)$$

   por lo tanto esta expresión es la invariante que se cumple durante la ejecución de todo el programa.

4.3.5. Ejercicios propuestos

1. Determinar la invariante del ciclo siguiente:

```
X=0;
while (X < 20){
  X=X+5;
}
```

2. Considere un programa de búsqueda secuencial que devuelve la posición del elemento buscado, o -1 cuando no existe. Cuál es la invariante apropiada?

```
int buscar(int [] v, int t){
// precondición el vector v no esta ordenado
// precondición t es el valor a buscar
int i=0;
while ((i<v.length()&&(v[i]!=t){
  i++;
 }
 if (i<n)
   return (i)
 else
   return (-1)
}
//post condición devuelve -1 cuando no existe y
//        devuelve la posición i cuando existe
```

3. Considere una clase que coloca y saca elementos en una estructura de pila. Escriba un programa e indique cuáles son las precondiciones, post condiciones e invariantes de la clase?

4. Probar que la invariante del procedimiento de división por restas sucecivas es $X = R + (YxQ)$

$$\{Y = y \wedge X = y\}$$
$$R = X;$$
$$Q = 0;$$
$$WHILE\ Y \leq R$$
$$R = R - Y; Q = Q + 1$$
$$\{R < y \wedge X = R + (YxQ)\}$$

4.4. Aplicaciones de las invariantes

Las aplicaciones de las invariantes se orientan basicamente en el desarrollo y verificación de los programas. Durante el proceso, el de verificar que las propiedades se cumplen durante la ejecución. En la verificación están los métodos formales que vienen de la lógica formal y de aquellos procedentes del cálculo de predicados [Red90].

Lo que se presenta es el desarrollo de programas utilizando los principios de verificación para, posteriormente introducirnos a la temática de verificar el funcionamiento del programa en tiempo de ejecución.

El método que se sigue consiste de los siguientes pasos:

1. Especificar el problema.
2. Escribir las pre y post condiciones .
3. Identificar las variables del problema.
4. Encontrar la invariante .
5. Codificar el ciclo, manteniendo la invariante en cada instante.
6. Prueba del programa.

Desarrollemos como ejemplo, el proceso de dividir un número por otro hallando el residuo, utilizando restas sucesivas.

Planteamiento del problema.- Dados dos número X, Y hallar Q denominado cociente y R denominado residuo tal que Q sea el cociente de la división y R sea el residuo.

Comenzamos escribiendo las pre y post condiciones

```
/* Programa que realiza una división por restas sucesivas
 * Precondición: x toma un valor y la variable y es > 0
 * Post condición: x=y*q +r
 * Donde q es el cociente y
 *       r es el resto
*/
```

El proceso continúa identificando las variables del programa que, las codificamos directamente:

```
/* Programa que realiza una división por restas sucesivas
 * Precondición: x toma un valor entero
 * y la variable entera y es > 0
 * Post condición: x=y*q +r
 * Donde q es el cociente y
 *       r es el resto
*/
```

```
divide(int x,int y) {
int q,r;
.....
.....
}
```

En este momento identificamos las características y las invariantes del problema utilizando las variables definidas en la pre y post condición . Primero el algoritmo consiste en restar sucecivamente el valor del divisor hasta hallar el cociente y el resto. Segundo observamos que el resto debe ser menor que el dividendo caso contrario se puede seguir restando. Los valores iniciales de las variables q, r serán $q = 0$, y $r = y$ respectivamente.

Codificando estas consideraciones

```
/* Programa que realiza una división por restas sucesivas
 * Precondición: x toma un valor entero
 * y la variable entera y es > 0
 * Post condición: x=y*q +r
 * Donde q es el cociente y
 *       r es el resto
*/
divide(int x,int y) {
int q=0;
int r=y;
while (r > = x) {
.....
.....
}
.....
.....
}
```

La invariante del ciclo debe mantenerse durante toda el ciclo. En nuestro caso es obvio que la invariante es $x = q \cdot y + r$

```
/* Programa que realiza una división por restas sucesivas
 * Precondición: x toma un valor entero
 * y la variable entera y es > 0
```

```
 * Post condición: x=y*q +r
 * Donde q es el cociente y
 *       r es el resto
*/
divide(int x,int y) {
int q=0;
int r=y;
while (r > = x) {
  //invariante x=y*q +r
r=r-y;
q++;
  }
}
```

reacomodando los comentarios tenemos la versión final.

```
/* Programa que realiza una división por restas sucesivas
 * Precondición: x toma un valor entero
 * y la variable entera y es > 0
 */
divide(int x,int y) {
int q=0;
int r=y;
while (r > = x) {
  //invariante x=y*q +r
r=r-y;
q++;
  }
 /* Post condición: x=y*q +r
 * Donde q es el cociente y
 *       r es el resto
*/
}
```

La invariante se verifica que es matemáticamente correcta. Tomando un ejemplo de ejecución para un caso particular se puede ver que la invariante no cambia. Vemos por ejemplo $9/2$:

Iteración	x	y	q	r	x=y*q +r
0	9	2	0	9	9
1	9	2	1	7	9
2	9	2	2	5	9
3	9	2	3	3	9
4	9	2	4	1	9

Como se observa en cada iteración se ha mantenido la invariante.

Veamos un ejemplo que utiliza dos bucles. Especifiquemos el programa para construir la criba de Eratostenes. Lo que se quiere es un vector donde los todos los múltiplos de algún número estén marcados. Los que no son múltiplos son números primos y no están marcados.

De aquí se ve que, la precondición del problema es un vector que tiene dos estados: marcado y no marcado. Los marcados son los números no primos. Al tener solo dos estados se puede utilizar un vector de bits. Escribimos la precondición, post condición así como, los elementos requeridos en el código.

```
import java.util.*;
/**
 * Programa para construir los numeros primos
 *    hasta = 100.000.000
 * utilizando la Criba de Eratostenes
 * Precondición:
 *    Un vector de bits con todos sus valores en cero
 */
public class Criba {
  public static void main(String[] s) {
    int n = 100000000;
    BitSet a = new BitSet(n + 1);
}
}
/**
 * Post condición:
 *    Un vector de bits con los numeros primos en cero.
 */
```

Ahora debemos construir las invariantes del ciclo. Recordemos que lo que se hace en el algoritmo es marcar todos los múltiplos de $2, 3, 4, 5, ...$ hasta la

raíz cuadrada del número máximo. Como se requieren dos ciclos también se requieren dos invariantes.

El resultado obtenido es:

```
import java.util.*;
/**
 * Programa para construir los numeros primos
 *    hasta = 100.000.000
 * utilizando la Criba de Eratostenes
 * Precondición:
 *   Un vector de bits con todos sus valores en cero
 */
public class Criba {
  public static void main(String[] s) {
    int n = 100000000;
    BitSet a = new BitSet(n + 1);
    int i = 2, j = 0;
    // construcción de la criba
    for (i = 2; i * i <= n; i++) {
    //Invariante hasta i-1  se han marcado todos los
    //multiplos de i - 1 para i > 2
      if (!a.get(i)) {
        for (j = i + i; j <= n; j=j+i;) {
        //Invariante j es multiplo de i
          a.set(j);
        }
      }
    }
}
}
/**
 * Post condición:
  *   Un vector de bits con los numeros primos en cero.
 */
```

Ahora completemos con las pre y post condiciones de cada ciclo. Las correspondientes al primer ciclo son las mismas del programa. Las corres-

pondientes al segundo ciclo son: Precondición: Los múltiplos de i no han sido marcados. Post condición: Los múltiplos de i han sido marcados. Introduciendo las mismas tenemos la versión final.

```
import java.util.*;
/**
 * Programa para construir los numeros primos
 * hasta = 100,000,000
 * utilizando la Criba de Eratostenes
 * Precondición:
 *   Un vector de bits con todos sus valores en cero
 */
public class Criba {
  public static void main(String[] s) {
    int n = 100000000;
    BitSet a = new BitSet(n + 1);
    int i = 2, j = 0;
    // construcción de la criba
    for (i = 2; i * i <= n; i++) {
    //Invariante hasta i-1  se han marcado todos los
    //multiplos de i - 1 para i > 2
      if (!a.get(i)) {
        //Precondición:
        //  los multiplos de i no han sido marcados
        for (j = i + i; j <= n; j=j+i;) {
        //Invariante j es multiplo de i
          a.set(j);
        }
        //Post condición:
        //los multiplos de i han sido marcados
      }
    }
}
}
/**
 * Post condición:
 //Un vector de bits con los numeros primos en cero.
 */
```

Una de las aplicaciones de las invariantes es caracterizar el desarrollo de un programa. El siguiente ejemplo muestra el desarrollo de un programa que evalúa la serie de Fibonacci que, como verá es difícil de comprender si no se sigue el desarrollo.

La serie de Fibonacci es formada por la ecuación $t(n)+t(n-1)$ con $t(0) = 1$. Un programa para evaluar esta secuencia de números sería el siguiente:

```
//Precondicion: N>=0
  y=1;
  n=0;
  x=0;
  while(n!=N){
     suma=x+y;
     x=y;
     y=suma;
     n++;
  }
//Post condicion y=Fibonacci(N)
```

Para mejorar el tiempo de proceso de este algoritmo, reemplacemos *suma* para obtener $y = x + y$, y luego, reescribimos las siguientes líneas de código: $x = y$ y $y = x + y$ como una ecuación matricial:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.4.1}$$

Denominando estas matrices A, B tenemos

$$A = \begin{pmatrix} x \\ y \end{pmatrix} B = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{4.4.2}$$

Si reemplazamos 4.4.2 en el programa la parte del ciclo se convierte en:

```
  while(n!=N){
     A=B*A
     n++;
  }
```

donde A y B son matrices. Analizando vemos que lo que se está calculando es $A = B^n A$. De aquí se puede ver que la invariante del ciclo es:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{4.4.3}$$

Se ve que lo que tenemos es una potencia de una matriz. Con esta idea podemos aplicar el método que mostramos para hallar una potencia en tiempo logarítmico.

Inicialmente hallemos las siguientes potencias:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \tag{4.4.4}$$

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}^4 = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix} \tag{4.4.5}$$

Esto nos lleva al hecho que, no demostraremos, de que todas las potencias pares tienen la forma:

$$\begin{pmatrix} a & b \\ b & a+b \end{pmatrix} \tag{4.4.6}$$

Calculando $B \cdot B$ se tiene que:

$$a = a \cdot a + b \cdot b$$

$$b = b \cdot a + a \cdot b + b \cdot b.$$

Calculando $A \cdot B$ se tiene que:

$$x = a \cdot x + b \cdot y\text{y}$$

$$y = b \cdot x + a \cdot y + b \cdot y.$$

Recordando que el algoritmo mejorado a $O(\log(n))$ para hallar las potencias trabaja bajo el siguiente método:

```
while(n!=N){
   si par(n)
  B=B*B
  n=n/2
   si impar(n)
      A=A*B
      n=n-1
}
```

Llevando ésto al código se tiene el programa final:

```
public class Fibonacci {
  /**
   * Programa para hallar el Fibonacci 8,n=8
   * @author Jorge Teran
   * @param args
   *            none
   */

  public static void main(String[] args) {
    int n=8,a=0,b=1,x=0,y=1,temp=0;
    while(n!=0){
      if(n%2==0){
        temp = a*a + b*b;
        b = b*a + a*b + b*b;
        a=temp;
        n=n/2;
      }

      else {
        temp = a*x + b*y;
        y = b*x + a*y + b*y;
        x=temp;
        n=n-1;
      }
    }
    System.out.println("x= "+x);
    System.out.println("y= "+y);
  }
}
```

Como se ve en la ejecución del programa $x = Fibonacci(n-1)$ y $y = Fibonacci(n)$. En este programa se ve claramente que la especificación de la invariante es esencial para entender el programa final.

4.4.1. Principio de correctitud

El principio de correctitud se define como:

- *inicialización* La invariante es verdadera cuando se ejecuta el programa por primera vez.
- *preservación* la invariante se preserva al principio, al final de cada uno de los ciclos del programa y cuando este termina.
- *terminación* El programa termina y produce los resultados esperados. Esto puede probarse utilizando los hechos que hacen la invariante.

El proceso de prueba moderno consiste en verificar el principio de correctitud. Puede realizarse, en forma estática (lógica formal) y en forma dinámica durante la realización del código.

4.5. Diseño por contratos

El diseño por contratos tiene sus orígenes en 1990 con los trabajos de Bertrand Meyer y el lenguaje de programación Eifel. Consiste en establecer relaciones cuyo fundamento están en la lógica formal entre las diferentes clases del programa.

La motivación principal para desarrollar el diseño por contratos es la reutilización del código. El diseño por contratos es un método de la ingeniería de software para la fiabilidad y aplicable a la programación orientada a objetos.

La fiabilidad del software se puede ver como dos aspectos: primero la correctitud y segundo la robustez.

Para describir la correctitud, supongamos que le traen 100,000 líneas de código y le preguntan si es correcto o no. Claramente no se puede decir mucho al respecto, lo que se requiere es una especificación detallada para verificar que el producto satisface la especificación.

Un elemento de software no es correcto o incorrecto por sí mismo. Es correcto o incorrecto con respecto a su especificación.

Definimos la correctidud como una tupla de pares de elementos:

$$correctitud = (especificacion, codigo).$$

La robustez es la facilidad que tiene un programa para reaccionar en forma razonable a condiciones que no están en la especificación.

En la construcción de programas existen dos metodologías principales que, son la *programación defensiva* y *programación por contratos.*

La programación defensiva se basa en ir agregando controles a fin de verificar posibles errores durante la ejecución del programa. En este tipo de programación se tiene cada vez más código, haciendo que el código nuevo requiera introducir nuevos controles.

Supóngase que se tiene una rutina que devuelve el resultado de dividir a/b el código podría ser

```
int divide (int a, int b){
    return a/b;
}
```

Aplicando la programación defensiva se tendría que codificar los posibles errores, por ejemplo $b > 0$ o si a puede estar en un rango de valores y así sucecivamente todas las posibles situaciones que puedan ocurrir a fin de que, la respuesta sea adecuada.

```
int Divide (int a, int b){
    if (!(b>0)){
        System.out.println("error b=0")
       }
 ......
 ......
 ......
   return a/b;
 }
```

Como se ve el concepto de una simple división ha sido complicada con una serie de operaciones que ocultan el próposito de la rutina *Divide*.

La programación por contratos toma los conceptos de la lógica formal y la especificación de programas. Esta metodología de diseño establece una relación contractual entre clases especificando los requerimientos y los resultados esperados. Esto se realiza incluyendo precondiciones, post condiciones e invariantes.

Se puede ver como un contrato entre personas donde cada una de las partes tiene una obligación que cumplir. En el ejemplo que vimos no es obligación de *Divide* que, los valores que recibe sean correctos, más bien el de dividir correctamente. La obligación del programa que llama a *Divide* es el de pasar los datos correctos.

Normalmente utilizamos la palabra *bug* para referirnos a cualquier tipo de problema de software. Esta palabra es un poco ambigua por lo que hay que definir los problemas que existen.

Un producto de software puede tener tres tipos de problemas: errores, defectos y fallos. Error es una mala decisión realizada durante el desarrollo del software. Un defecto es una propiedad que hace que no se adecue a su especificación. Un fallo es un evento incorrecto que, en su comportamiento esperado, ocurre en alguna de sus ejecuciones.

La programación por contratos divide en obligaciones y beneficios entre proveedor y cliente.

Cliente Es un programa que hace uso de un objeto (programa) conoce la interfaz de la clase y no sabe nada de la implementación

Proveedor Es un programa (objeto) que es utilizado por un cliente, construye la documentación, la mantiene y publica una interfaz.

El siguiente cuadro muestra los beneficios y obligaciones de clientes y proveedores:

	Obligaciones	Beneficios
Cliente	Satisfacer la precondición	Obtener los resultados de la post condición
Proveedor	Satisfacer la post condición	Un proceso más simple gracias a que se presume que se satisface la precondición

En nuestro caso el programa que llama a la rutina de dividir debe garantizar que los datos enviados son correctos y el programa que realiza la división garantiza que la respuesta es correcta.

En el enfoque tradicional se presentan varios problemas:

- El cliente no entiende en que situaciones puede utilizar una clase.
- El cliente no entiende las consecuencias de utilizar un método.
- Cuando se cambia un código, ¿Cómo se consigue que la interfaz se mantenga?

- Si la interfaz cambia, ¿Cómo se entera el cliente?

Con esta metodología es muy fácil determinar la causa de un fallo en un programa:

- Una falla en la precondición es una manifestación de una falla en el cliente
- Una falla en la post condición es una manifestación de una falla en el proveedor.

En un contrato entre cliente y proveedor se pueden dar dos situaciones, la primera es que el contrato se cumpla con lo cual no hay que hacer absolutamente nada, o puede que, el contrato se rompa en cuyo caso se puede utilizar una clase especializada para manejar el error o simplemente terminar el proceso indicando la causa de la falla.

En todo el proceso de diseño por contratos se debe evitar la redundancia. Esta regla es el opuesto de la programación defensiva. No está por demás realizar controles en el código, sin embargo, pueden hacer el código muy poco entendible. En el diseño bajo contratos se debe colocar estos controles en las precondiciones.

El objetivo es la simplicidad, al agregar código redundante en la programación defensiva, se hace el software más complicado. Al agregar más código hay que agregar más controles y así sucesivamente.

4.5.1. Especificación de contratos

Los contratos se especifican a través de claúsulas que definen las pre y post condiciones. Las precondiciones se especifican con la claúsula *requires* y las post condiciones con la claúsula *ensures*.

Supongamos que se quiere especificar una clase *Tiempo* que tiene los métodos *ponerHora*, *ponerMinutos*, *ponerSegundos*, *tiempoTranscurrido*.

Una posible implementación puede ser crear tres variables *hora*, *minutos*, *segundos* y a través de un algoritmo hallar el tiempo transcurrido.

Otra implementación puede ser hallar el tiempo transcurrido desde un momento determinado, por ejemplo, el principio del año. De esta forma es posible almacenar el tiempo en una sola variable entera. Con este método se puede simplificar el hallar el tiempo transcurrido a restar dos variables enteras.

Con el sistema de diseño por contrato solo se debe especificar el contrato que las clases deben cumplir y no la implementación. La clase *ponerHora* se puede especificar como sigue:

```
void ponerHora(int h)
/* requieres 0 <= h <= 23
 * ensures hora=h
 * ensures old minutos = minutos
 * ensures old segundos = segundos
*/
```

Esta especificación garantiza que la hora está en un rango permitido, que los minutos y segundos no cambian y que la variable *h* toma el valor de *hora*. Una implementación de esta especificación podría ser

```
void ponerHora(int h)
/* requieres 0 <= h <= 23
 * ensures hora=h
 * ensures minutos = minutos
 * ensures segundos = segundos
*/
  hora=h;
}
```

Si analizamos el código propuesto, parece que la post condición es una redundancia del código. Hay que hacer notar que la post condición *hora=h* indica que lo que se espera, es que la variable *hora* tome el valor *h*. El código indica como se hace esto.

Si se hubiera escogido la segunda forma de implementar, se tendría un código más complejo y la misma especificación. Esto permite que podamos cambiar el código sin tener que avisar este cambio a todos los clientes, facilitando la reusabilidad y mantenimiento al haber mantenido los contratos entre clientes y proveedores.

Cada pre y post condición debe satisfacer los siguientes requerimientos:

- Las pre y post condición aparece en la documentación oficial distribuida a autores de módulos cliente.
- Es posible justificar la precondición en base de la especificación solamente.

- Cada característica que aparece en la pre y post condición debe estar disponible a cada cliente cuya ejecución esté disponible (regla de disponibilidad).

4.5.2. Invariantes

En el diseño por contratos se tienen dos tipos de invariantes, las invariantes de clase y las invariantes de bucles o ciclos.

Las invariantes de clase describen propiedades globales que deben preservarse entre todas las rutinas. En cambio las invariantes de bucle describen las propiedades que deben preservarse entre las variables de un bucle.

Una invariante es correcta si y solo si cumple:

- La creación de un procedimiento p, cuando se aplica a los argumentos que satisfacen la precondición, satisface la invariante *I*.
- Cuando termina una rutina exporta sus resultados, cuando satisface la precondición dando un estado *I*, también satisface la invariante. Termina en un estado que satisface la invariante

En la clase *Tiempo* la invariante quedaría especificada como:

```
Tiempo (){
    Invariant 0 < = hora < =24
    Invariant 0 < = minutos < =59
    Invariant 0 < = segundos < =59
}
```

El estado global de la clase especificada indica que cuando cambiamos la hora, los valores hora, minutos y segundos están en el rango correcto.

Una especificación de la clase tiempo puede ser como sigue:

```
public clase Tiempo (){
    Invariant 0 < = hora < =24
    Invariant 0 < = minutos < =59
    Invariant 0 < = segundos < =59

ponerHora(int h){
}
```

```
/* requieres 0 <= h <= 23
 * ensures hora=h
 * ensures minutos = minutos
 * ensures segundos = segundos
*/
}
ponerMinutos(int m){
}
/* requieres 0 <= h <= 23
 * ensures minutos=m
 * ensures hora = hora
 * ensures segundos = segundos
*/
......
........
```

En cada método se deberá especificar las invariantes del ciclo apropiadas en función de la implementación.

Como se ve estos conceptos simplifican la construcción del programa y lo hacen más legible. En Java se puede implementar a través de varios programas utilitarios que, se muestran en la siguiente sección.

4.6. Prueba estática

Las pruebas estáticas de código utilizan la lógica formal para probar que aplicando a la precondición la invariante se llega a la post condición. Estas pruebas se realizan aplicando una serie de reglas de la lógica formal. El propósito de presentar estas reglas es mostrar en forma simple como se aplican en el proceso de prueba. El texto de la Dra. Tanja Vos [Vos01], provee la prueba y un extensa explicación de esta temática.

A fin de ejemplificar el proceso que se sigue veamos un ejemplo muy sencillo:

La regla de bloques indica que dada una precondición $\{P\}$, con un código C se llega a la post condición $\{Q\}$. Las variables locales no deben figurar en la precondición y post condición. Ejemplo:

$$\{X = x\}X = X + 1\{X = x + 1\}$$

El procedimiento de prueba consiste en reemplazar las variables de la precondición en el código para obtener la post condición. En el ejemplo reemplazamos valor de la variable X obteniendo:

$$\{X = x\}x = x + 1\{X = x + 1\}$$

finalmente reemplazamos la x y se obtiene la post condición.

En el caso de las secuencias se procede de forma similar Dada una secuencia de instrucciones

$$C_1; C_2; ...C_n$$

una precondición $\{P\}$ y la post condición $\{Q\}$, ésta secuencia de instrucciones puede expresarse como:

$$\{P\}$$

$$\{P_1\}C_1; \{Q_1\}\{P_2\}C_2; \{Q_2\}.....\{P_n\}C_n; \{Q_n\}$$

$$\{Q\}$$

De este desarrollo se ve que P_1 es consecuencia de P y que P_2 es consecuencia de Q_1 y sucecivamente hasta llegar a Q que es consecuencia de Q_n.

Para este proceso existen reglas, para secuencias, bloques, condicionales, bucles while, for y todos les elementos utilizados en la programación.

4.7. Prueba dinámica

La prueba dinámica es la que se realiza en tiempo de ejecución y el lenguaje de programación brinda el método de afirmaciones a través de la instrucción *assert*. Con aplicaciones adicionales se pueden verificar las precondiciones, post condiciones e invariantes. Estos controles son insertados en el código automáticamente con afirmaciones. En estos desarrollos se encuentra *iContract*, $JMSAssert$ [Fel05], y Java Modeling Language (JML).

4.7.1. Afirmaciones

Las afirmaciones son una forma de verificar en tiempo de ejecución, ciertas condiciones que se deben cumplir durante la ejecución del programa. Esta opción viene en Java después de la versión 1.4. La utilización dentro del

lenguaje Java se realiza con la instrucción *assert* que tiene las dos formas siguientes:

$$assert\ expresion1;$$

$$assert\ expresion1\ : expresion2;$$

En la primera forma $expresion1$ es una expresión booleana que el programador afirma ser verdadera durante la ejecución del programa.

En la segunda forma se agrega $expresion2$ que es un mecanismo para pasar una cadena que será mostrada en caso de fallar la afirmación.

Cuando uno ejecuta un programa en modo de prueba está interesado en que se procesen las afirmaciones, sin embargo, cuando está en modo de producción se desea eliminar estas instrucciones para mejorar el tiempo de proceso. Por este motivo al invocar la ejecución del programa se debe incluir la opción $-ea$ para que se ejecuten. Por ejemplo se queremos ejecutar el programa *Ejemplo* con las afirmaciones habilitadas se coloca

$$java\ -ea\ Ejemplo$$

El propio lenguaje Java trae la opción que permite habilitar las afirmaciones propias del lenguaje a fin de diagnosticar posibles errores propios del Java, esto se hace con la opción $-esa$. Recordando el ejemplo de la sección anterior:

```
Nombre persona = new Nombre(nombrePersona);

int largo=0;
String dato = persona.devuelveNombre();

   if (dato == null) {
       System.err.println("Error grave:
          Falta la propiedad 'nombre' ");
      }
      else
      {
        largo=dato.trim().length();
      }
```

Se observa que el control sobre el nombre se hace permanentemente. Cuando el sistema está completamente depurado es muy probable que no se requiera realizar este control. Utilizando la instrucción *assert* queda como sigue:

```
Nombre persona = new Nombre(nombrePersona);

int largo=0;
String dato = persona.devuelveNombre();

assert dato == null : "Falta la propiedad 'nombre' "
largo=dato.trim().length();
```

Como puede observar el programa ha quedado en formas más comprensible y la cantidad de código a revisar es menor, pudiendo habilitar o desabilitar el control en tiempo de ejecución.

Retomando el ejemplo de la criba de Eratostenes en versión final. Reescribimos algunas invariantes y precondiciones en una instrucción *assert* para obtener:

```
import java.util.*;

public class Criba {
    public static void main(String[] s) {
        int n = 100;
        BitSet a = new BitSet(n + 1);
        int i = 2, j = 0;
        for (i = 2; i * i <= n; i++) {
            // Invariante hasta i-1 se han marcado todos los
            // multiplos de i - 1 para i > 2
            if (!a.get(i)) {
                // Precondición:
                // los multiplos de i no han sido marcados
                assert !a.get(i) : "Procesando multiplos "
                        + i + " dos veces";
                for (j = i + i; j <= n; j = j + i) {
                    // Invariante j es multiplo de i
                    assert j % i == 0 :
                                " J no es multiplo de I";
                    a.set(j);
```

```
                }
            }
        }
    }
}
/**
 * Post condición:
 * Un vector de bits con los numeros primos en cero.
 */
```

Ahora en una ejecución normal del programa se ve que no hay ningún problema. Sin embargo, al codificar el código se puede haber escrito con un error:

```
for (j = i + i; j <= n; j++;) {
//Invariante j es multiplo de i
assert i%j == 0: " J no es multiplo de I"
```

Este error que puede pasar desapercibido por la costumbre de escribir el código de esa forma. En el tiempo de ejecución al habilitar las invariantes se obtiene el mensaje de error.

Esta forma de describir el código requiere que se busquen formas de especificar las pre, post condiciones e invariantes del programa.

Como se ve éstos conceptos simplifican la construcción del programa y lo hace más legible.

Capítulo 5

Aplicaciones de búsqueda y clasificación

5.1. Introducción

La búsqueda y la clasificación tienen una serie de aplicaciones que serán descritas en las siguientes secciones. Se procederá, de métodos generales a casos particulares que reducen el tiempo de proceso. El proceso de especificación se realiza como se mostró en el capítulo anterior especificando las invariantes, pre y post condiciones sin entrar en la demostración de éstas. Los libros de Horstman [CSH05] proporcionan una buena descripción de las librerías de Java.

5.2. Algoritmos de búsqueda

La búsqueda de elementos ha sido analizada en diferentes entornos, memoria, bases de datos, texto plano y otros. Cada algoritmo de búsqueda da como resultado una eficiencia diferente en función de como se ha organizado la información. La búsqueda en forma genérica se presenta como, el mecanismo de hallar un elemento en un vector del cual solo conocemos, el número de elementos que contiene. Este algoritmo, denominado búsqueda secuencial, viene especificado como sigue:

```
public class Sec {
  /* Programa de busqueda secuencial
```

```
   *@ author Jorge Teran
   */
    static int buscar(int[] v, int t) {
    /*@ requires v != null ;
      @ requires t != null ;
      @ ensures  (hallo = -1)
      @  || (hallo < v.length && hallo >=0);
      @*/
    int hallo = -1;
//    @ loop_invariant i < v.length && hallo = -1;
    for (int i = 0; i < v.length; i++)
      if (v[i] == t) {
        hallo = i;
        break;
       }
    return (hallo);
  }
```

Aplicando la teoría vista se determina que el tiempo de ejecución del algoritmo es proporcional

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

Esto significa que es necesario conocer más sobre el conjunto de datos para mejorar el tiempo de búsqueda. Este concepto hace que se puedan realizar búsquedas específicas para cada conjunto de datos a fin de mejorar el tiempo de proceso.

Supóngase que se tiene una lista de números enteros entre 0 y n. Estos números corresponden al código de producto en un almacén. Dado un número se quiere conocer si este pertenece al almacén. Para esto podemos organizar los números de varia formas. Se puede colocar la información en un vector y proceder con el esquema de búsqueda anterior obteniento un tiempo $O(n) = n$. ¿Será posible obtener un tiempo $O(n) = 1$?.

La respuesta es sí. Consideremos un código n pequeño entonces podemos armar un vector de bits en memoria, qué indique que números existen y cuales no.

Se quiere cambiar la búsqueda secuencial aprovechando algunas características particulares al problema, suponiendo que los datos están previamente ordenados.

Esta solución debe funcionar tanto para enteros, cadenas y reales siempre y cuando todos los elementos sean del mismo tipo. La respuesta se almacena en un entero p que representa la posición en el arreglo donde se encuentra el elemento t que estamos buscando, -1 indica que el elemento buscado no existe en el arreglo.

Esta búsqueda denominada búsqueda binaria, resuelve el problema recordando permanentemente el rango en el cual el arreglo almacena t. Inicialmente el rango es todo el arreglo y luego es reducido comparando t con el valor del medio y descartando una mitad. El proceso termina cuando se halla el valor de t o el rango es vacío. En una tabla de n elementos en el peor caso realiza $\log_2 n$ comparaciones.

La idea principal es que t debe estar en el rango del vector. Se utiliza la descripción para construir el programa.

```
precondición el vector esta ordenado
inicializar el rango entre $0..n-1$
loop
    {invariante: el valor a buscar debe estar en el rango}
    si el rango es vacio
        terminar y avisar que t no esta en el arreglo
    calcular m que es el medio del rango
    use m con una prueba para reducir el rango
    si se encuentra t en el proceso de reducción
        terminar y comunicar su posición
```

La parte esencial del programa es la invariante que al principio y final de cada iteración nos permite tener el estado del programa y formalizar la noción intuitiva que teníamos.

Se construye el programa utilizando refinamiento sucesivo haciendo que todos los pasos respeten la invariante del mismo.

Primero buscaremos una representación del rango digamos $l..u$ entonces la invariante es *el rango debe estar entre* $l..u$. El próximo paso es la inicialización y debemos estar seguros que respeten la invariante, la elección obvia es $l = 0, u = n - 1$.

Codificando un programa con los conceptos detallados previamente tenemos:

```
precondición x debe estar ordenado
post condición p especifica la posición
```

```
                o p es -1 cuando no existe
    l=0; u=n-1
    loop
        {invariante: debe estar en el rango l,u}
    si el rango es vacio
        terminar y avisar que t no esta en el arreglo
    calcular m que es el medio del rango
    use m con una prueba para reducir el rango
    si se encuentra t en el proceso de reducción
        terminar y comunicar su posición
```

Continuando con el programa vemos que el rango es vacío cuando $l > u$. en esta situación terminamos y devolvemos $p = -1$. Llevando esto al programa tenemos:

```
precondición x debe estar ordenado
    post condición p especifica la posición
                o p es -1 cuando no existe
    l=0; u=n-1
    loop
        {invariante: debe estar en el rango l,u}
        if l > u
        p=-1; break;
    calcular m que es el medio del rango
    use m con una prueba para reducir el rango
    si se encuentra t en el proceso de reducción
        terminar y comunicar su posición
```

Ahora calculamos m con $m = (l + u)/2$ donde / implementa la división entera. Las siguientes líneas implican el comparar t con $x[m]$ en la que hay tres posibilidades, por igual, menor y mayor. Con lo que el programa queda como sigue:

```
    precondición x debe estar ordenado
    post condición p especifica la posición
                o p es -1 cuando no existe
    l=0; u=n-1
    loop
        {invariante: debe estar en el rango l,u}
```

```
if l > u
p=-1; break;
m=(l+u)/2;
case
    x[m]< t : l=m+1;
    x[m]= t : p=m: break;
    x[m]> t : u=m-1
```

Se deja como ejercicio para el lector comprobar que el tiempo de ejecución de este algoritmo es proporcional a $\log(n)$. Para resolver esto fácilmente se puede reescribir en forma recursiva y aplicar el teorema master.

5.2.1. Prueba exhaustiva

Ahora es necesario probar la búsqueda binaria. El primer paso es el de convertir la invariante, pre y post condiciones con afirmaciones o utilizando el lenguaje JML. Sin embargo, esto no garantiza que el programa funcione correctamente para todos los casos.

Consideremos el siguiente pseudo código y realicemos una prueba de escritorio:

```
public class búsqueda {

  public static int BusquedaBinaria(int[] x, int t) {
    int l = 0;
    int u = x.length - 1;
    int m = 0;
    int p = -1;
    while (l <= u) {
      m = (l + u) / 2;
      if (x[m] < t)
         l= m ;
      else if (x[m] == t) {
         p = m;
         break;
      } else
         u = m;
    }
    return p;
```

```
 }
```

Aparentemente es correcto, pero una prueba de escritorio extensiva es larga y aburrida. Por esto es conveniente construir un conjunto de datos de prueba, para lo cual, utilizamos el siguiente código:

```
int[] x = new int[MAXIMO];
for (int i = 0; i < MAXIMO; i++) {
   x[i] = i * 5;
 }
```

En este conjunto de datos conocemos perfectamente el contenido del vector, vale decir que, si buscamos un múltiplo de 5 debería encontrarlo.

Un programa puede estar *probado* o *certificado*. Decimos que esta probado cuando funciona correctamente con los datos de prueba utilizados.

Decimos que el programa está certificado cuando funciona correctamente con todos los posibles casos que pueden ocurrir.

Por ejemplo, con el vector de datos construidos podemos probar con algunos valores y ver que funciona adecuadamente. ¿Podremos decir lo mismo para todo los valores?

Para realizar esta prueba vamos a buscar todos los elementos introducidos al vector y verificar que los encuentra y devuelve el resultado correcto. El código es el siguiente:

```
for (int i = 0; i < MAXIMO; i++) {
   if (búsqueda.BusquedaBinaria(x, i * 5) != i) {
       System.out.println("Error al buscar " + i*5);
      }
```

Ahora sabemos que se prueba la búsqueda de todos los valores existentes en el vector y no produce errores.

Los valores que no existen en el vector pueden pertenecer a cualquier rango de datos. Por esto se hace necesario buscar un elemento no existente entre cada uno de los valores introducidos obteniendo el sigiente código:

```
for (int i = 0; i < MAXIMO; i++) {
     if (búsqueda.BusquedaBinaria(x, i * 5 + 1) != -1) {
        System.out.println("Error al buscar " + i*5+1);
     }
```

Este código nos certifica que el programa es correcto en un determinado conjunto de datos. Que podemos decir si ampliamos el tamaño del vector? Solo que se ha probado y puede que funcione pero, no está certificado. El código completo es:

```
public static final int MAXIMO = 1000;

public static void main(String[] args) {

   System.out.println("Creando el vector");
   int[] x = new int[MAXIMO];
   for (int i = 0; i < MAXIMO; i++) {
       x[i] = i * 5;
      }
   System.out.println("Buscando elementos existentes");
   for (int i = 0; i < MAXIMO; i++) {
      if (búsqueda.BusquedaBinaria(x, i * 5) != i) {
         System.out.println("Error al buscar " + i*5);
      }
   }
   System.out.println("Buscando elementos no existentes");
   for (int i = 0; i < MAXIMO; i++) {
      if (búsqueda.BusquedaBinaria(x, i * 5 + 1) != -1) {
         System.out.println("Error al buscar " + i*5+1);
      }
   }
   System.out.println("Fin de la prueba");
}
```

Nos preguntamos que es lo que se debe especificar. Generalmente se deben colocar las invariantes, pre y post condiciones. Sin embargo es útil también afirmar o especificar el $O(n)$ para verificar el funcionamiento del programa en los tiempos previstos o detectar bucles infinitos.

5.2.2. Representación gráfica de la búsqueda

Una representación gráfica puede ser adecuada para mostrar como funciona un programa. Esto hace que de un golpe de vista se determine que el programa funcione, aparentemente sin errores.

Para realizar una representación gráfica del programa de búsqueda binaria construimos una línea para mostrar el rango de búsqueda. El programa debe ser capaz de mostrar como se reduce el rango y contar el número de iteraciones del mismo.

Para ejemplificar ésto se muestra un programa que construye un vector con los números pares entre 0 y 60. Luego permite buscar un valor mostrando el proceso seguido en la búsqueda.

```
/*
 * Programa para ejemplificar graficamente la invariante
 * de la busqueda binaria
 * @author Jorge Teran
 */
 import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.event.*;

public class Busbin {
  public static void main(String[] args) {
    TextFrame frame = new TextFrame();
    frame.setSize(640,480);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}

class TextFrame extends JFrame
{
public TextFrame(){
setTitle("búsqueda Binaria");

DocumentListener listener = new TextoFieldListener();

// agregar un panel con el texto a buscar

JPanel panel = new JPanel();
```

```
panel.add(new JLabel("Buscar:"));
queBuscaField = new JTextField("30", 5);
panel.add(queBuscaField);
queBuscaField.getDocument().addDocumentListener(listener);

add(panel, BorderLayout.NORTH);

// agregar el grafico
muestra = new DrawPanel();
add(muestra, BorderLayout.CENTER);
pack();
}
// recuperar el valor a buscar
public void setTexto(){
try {
    int queBuscas = Integer.parseInt(
                    queBuscaField.getText().trim());
    muestra.muestra(queBuscas);
     }
catch (NumberFormatException e) {}
//no se reliza nada si no se puede interpretar los datos
     }

public static final int DEFAULT_WIDTH = 640;
public static final int DEFAULT_HEIGHT = 480;

private JTextField queBuscaField;
private DrawPanel muestra;

private class TextoFieldListener
              implements DocumentListener {
  public void insertUpdate(DocumentEvent event)
              {setTexto();}
  public void removeUpdate(DocumentEvent event)
              {setTexto();}
  public void changedUpdate(DocumentEvent event) {}
}
```

```
}

class DrawPanel extends JPanel {
  int t=30, incrementos=20,
    medio=0,maximo=30;
  int[] v = new int[maximo + 1];
  public DrawPanel(){
  for (int i = 0; i <= maximo; i++) {
    v[i] = i * 2;
    }
}
public void muestra(int x){
    t=x;
     repaint();
}
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  int desde=20, topY=100,
      hasta=600,l=0,u=maximo,m=0;
  Graphics2D g2 = (Graphics2D) g;
  g2.setPaint(Color.WHITE);
  g2.fillRect(0,0,640,480);
  g2.setColor(Color.BLACK);
  int hallo=-1,contador=0;
// dibujar
  String explica="Las lineas representan el "+
              "rango en el cual se busca";
  g.drawString(explica,
    ((desde+hasta)/2 ) - (explica.length()*3), topY);
  while (l <= u) {
    topY += incrementos;
    m = (l + u) / 2;
    medio = (hasta + desde) / 2;
    g.drawString("" + contador++, 5, topY);
    g.drawString("" + v[l], desde, topY);
    g.drawString("" + v[u], hasta, topY);
    g.drawString("" + v[m], medio, topY);
    g2.draw(new Line2D.Double(desde, topY, hasta, topY));
```

Figura 5.1: Representación gráfica de la búsqueda binaria

```
    if (v[m] < t) {
        l = m + 1;
        desde = medio + 1;
      } else if (v[m] == t) {
        topY += incrementos;
        g.drawString("" + contador++, 5, topY);
        g.drawString("" + v[m],(hasta + desde)/ 2,topY);
        hallo = m;
        break;
      } else {
        u = m - 1;
        hasta = medio - 1;
      }
    }
  }
}
```

La salida del programa se ve en la figura 5.1 que, muestra como se va reduciendo el rango de búsqueda hasta encontrar el valor. A la izquierda se ve un contador con el número de iteración.

5.3. Clasificación

La clasificación o procedimiento de ordenar es uno de los problemas más importantes en la computación y existen un sin número de problemas de programación que requieren de ella. A continuación se detallan una serie de aplicaciones sin pretender que sean todas.

- Cómo podemos probar que en un grupo de elementos no existen elementos duplicados? Bien para esto ordenamos el conjunto y posteriormente verificamos que no existen valores tales que $x[i] = x[i+1]$ recorriendo el conjunto para todos los valores de i permitidos.

- Cómo podemos eliminar los duplicados? Ordenamos el conjunto de datos y luego utilizamos dos índices haciendo $x[j] = x[i]$. Se incrementa el valor de i y j en cada iteración. Cuando se halla un duplicado solo incrementa i. Al finalizar ajustamos el tamaño del conjunto al valor de j.

- Supongamos que tenemos una lista de trabajos que tenemos que realizar con una prioridad específica. Puede ser por ejemplo una cola de impresión donde se ordenan los trabajos en función de alguna prioridad. Para esto ordenamos los trabajos según la prioridad y luego se los procesa en este orden. Por supuesto que hay que tomar recaudos al agregar nuevos elementos y mantener la lista ordenada.

- Una técnica muy utilizada para agrupar ítems es la denominada cortes de control. Por ejemplo, si se desea obtener los totales de sueldo por departamento en una organización, ordenamos los datos por el departamento que, se convierte en el elemento de control y luego cada vez que cambia imprimimos un total. Esto se denomina un corte de control. Pueden existir varios cortes de control, por ejemplo sección, departamento, etc.

- Si deseamos hallar la mediana de un conjunto de datos podemos indicar que la mediana está en el medio. Por ejemplo para hallar el elemento k ésimo mayor bastaría en acceder a $x[k]$.

- Cuando queremos contar frecuencias y desconocemos el número de elementos existentes o este es muy grande, ordenando y aplicando el concepto de control, con un solo barrido podemos listar todos las frecuencias.

- Para realizar unión de conjuntos se pueden ordenar los conjuntos, realizar un proceso de intercalación. Cuando hay dos elementos iguales solo insertamos uno eliminando los duplicados.
- En el caso de la intersección intercalamos los dos conjuntos ordenados y dejamos uno solo de los que existan en ambos conjuntos.
- Para reconstruir el orden original es necessario crear un campo adicional que mantenga el original para realizar una nueva ordenación para, reconstruir el orden original.
- Como vimos también se pueden ordenar los datos para realizar búsquedas rápidas.

5.3.1. Clasificación en Java

Los métodos propios del Java proveen rutinas para ordenar objetos que facilitan el desarrollo de aplicaciones.

Para ordenar vectores que pueden ser de tipos int, long, short, char, byte, float o double es suficiente utilizar la clase *Arrays* con el método *sort*, el siguiente ejemplo genera 10 números al azar y los ordena.

```
import java.util.*;
/**
 * Programa ordenar enteros
 * utilizando el metodo de java
 * @author Jorge Teran
 */

public class SortVect {
  public static void main(String[] args) {
    int[] x = new int[10];
    Random gen = new Random();
//Generar 10 números enteros entre 0 y 100
    for (int i = 0; i < 10; i++)
      x[i] = gen.nextInt(100);
```

```
//Ordenar en forma ascendente
    Arrays.sort(x);
//mostrar los números ordenados
    for (int i = 0; i < 10; i++)
      System.out.println(x[i]);
  }
}
```

Cuando se requiere ordenar un objeto se hace necesario crear un mecanismo para comparar los elementos del objeto. Construyamos una clase *Persona* que almacene nombre, apellido y nota.

```
import java.util.*;

class Persona {
   private String nombre;
   private String apellido;
   private int nota;
   public Persona(String no, String ap, int n) {
      nombre = no;
      apellido = ap;
      nota = n;
   }

   public String getNombre() {
      return nombre;
   }
   public String getApellido() {
      return apellido;
   }

   public int getNota() {
      return nota;
   }
}
```

Para crear diferentes instancias de *Persona* se procede como sigue:

```
Persona[] alumnos = new Persona[3];
```

```
alumnos[0] = new Persona("Jose","Meriles", 70);
alumnos[1] = new Persona("Maria","Choque",55);
alumnos[2] = new Persona("Laura","Laura", 85);
```

Si se requiere ordenar éstos alumnos por nota no es posible utilizar directamente *Arrays.sort(alumnos)* dado que el método de clasificación no conoce cuáles y qué campos comparar para realizar el ordenamiento.

Por esto es necesario primeramente escribir un método que resuelva el problema. En nuestro ejemplo creamos el método *compareTo*. Este nombre es un nombre reservado de Java y lo que realiza es una sobrecarga de método reemplazando el método de Java con el nuestro.

Los valores que debe devolver son -1 cuando es menor, 0 cuando es igual y 1 cuando es mayor. Lo que hace este método es comparar el valor presente con otro pasado a la rutina. Esta rutina utiliza el método qsort para ordenar los valores. Esta rutina queda como sigue:

```
public int compareTo(Persona otro){
   if (nota < otro.nota) return -1;
   if (nota > otro.nota) return 1;
   return 0;
 }
```

Ademas es necesario especificar que la clase persona incluye el método *Comparable* y se especifica así:

```
class Persona implements Comparable<Persona>
```

El programa final queda implementado en el siguiente código:

```
import java.util.*;
/**
 * Programa ordenar objetos
 * utilizando el metodo de java
 * @author Jorge Teran
 */

public class OrdObjeto {
  public static void main(String[] args) {
    Persona[] alumnos = new Persona[3];
```

```
    alumnos[0] = new Persona("Jose", "Meriles", 70);
    alumnos[1] = new Persona("Maria", "Choque", 55);
    alumnos[2] = new Persona("Laura", "Laura", 85);
    Arrays.sort(alumnos);

    for (Persona e : alumnos)
      System.out.println("Nombre=" + e.getNombre()
          + ", Apellido= " + e.getApellido()
          + ", nota=" + e.getNota());
  }
}

class Persona implements Comparable<Persona> {

  private String nombre;

  private String apellido;

  private int nota;

  public Persona(String no, String ap, int n) {
    nombre = no;
    apellido = ap;
    nota = n;
  }

  public String getNombre() {
    return nombre;
  }

  public String getApellido() {
    return apellido;
  }

  public int getNota() {
    return nota;
  }
```

```
  public int compareTo(Persona otro) {
    if (nota < otro.nota)
      return -1;
    if (nota > otro.nota)
      return 1;
    return 0;
  }
}
```

5.3.2. Algoritmos de clasificación

Dado que el lenguaje ya incluye un soporte para realizar clasificaciones con un algoritmo parece que no es necesario revisar algorimos de clasificación. En la realidad existen problemas que pueden ser resueltos más eficientemente con un algoritmo específico. Esto se logra conociendo como están los datos que pretendemos procesar.

Los algoritmos para ordenar se clasifican en algoritmos generales y particulares. Los algoritmos generales se pueden aplicar sin importarnos como son los datos, en cambio los algoritmos particulares son aplicables a casos especiales para obtener un mejor tiempo de proceso.

Entre los algoritmos generales se explican los métodos de clasificación por inserción, selección, qsort, y el de burbuja.

Para entender los diferentes algoritmos de ordenar es necesario comprender como establecer la invariante en cada uno de los métodos. Para esto consideraremos que los datos a ordenar corresponden al eje y de un gráfico bidimencional y la posición del vector al eje x.

Con este gráfico los datos iniciales, es decir, la precondición vendría dada por el gráfico:

Una vez ordenados todos los elementos, el gráfico que representa a la postcondición sería el siguiente:

Método de la burbuja

El método de la burbuja es el más simple y el más antiguo. Tambien hay que mencionar que es el método más lento para ordenar.

El método consiste en comparar todos los elementos de la lista con el elemento de su lado. Si se requiere se realiza un intercambio para que estén en

el orden previsto. Este proceso se repite hasta que todos los elementos estén ordenados. Vale decir que, en alguna pasada no se realiza ningún intercambio.

El código para este algoritmo es el siguiente:

```
void Burbuja (int[] x) {
    int i, j, temp;
    int n=x.length;
    for (i = (n - 1); i >= 0; i--) {
      for (j = 1; j <= i; j++) {
        if (x[j - 1] > x[j]) {
          temp = x[j-1];
          x[j-1] = x[j];
          x[j] = temp;        }
      }
    }
```

Como se ve en el programa en el primer ciclo se recorren todos los elementos. El segundo ciclo se encarga de que el último elemento sea el mayor de todos.

Analizando el programa se puede determinar que, el tiempo de proceso en el caso general es:

$$\sum_{i=n-1}^{0} \sum_{j=1}^{j=i} = \sum_{i=n-1}^{0} i = n\frac{n-1}{2}$$

como se ve, demora un tiempo proporcional a $O(n^2)$ que lo hace impráctico. Podemos mencionar como una ventaja la simplicidad de la codificación y como desventaja la ineficiencia del algoritmo.

Existen otros métodos que también tienen un tiempo similar pero que son mucho más eficientes que mencionamos a continuación.

Clasificación por inserción

Cuando se tiene un conjunto de cartas en la mano lo que hacemos es tomar una carta y buscar su ubicación e insertarla en su sitio hasta ordenar todas. Para ésto podemos suponer que el primer elemento está en su lugar y proceder a colocar los siguientes en función del primer elemento.

Representando gráficamente el método obtenemos

La invariante indica que los valores hasta i están ordenados y los datos posteriores pueden ser menores o mayores que el último dato ya ordenado.

```
void Insercion(int[] x) {
  int i, j, temp;
  int n = x.length;
  for (i = 1; i < n; i++) {
  //los datos estan ordenados hasta i
    for (j = i; j > 0 && x[j - 1] > x[j]; j--) {
      temp = x[j-1];
      x[j-1] = x[j];
      x[j] = temp;
    }
  }
}
```

Para explicar el funcionamiento del algoritmo supongamos que inicialmente tenemos la siguiente secuencia de números

62 46 97 0 30

Inicialmente suponemos que el primer valor es el menor y lo insertamos en la primera posición que, es el lugar donde estaba inicialmente, luego se toma el siguiente valor y se ve si le corresponde estar antes o después, en el ejemplo se produce un intercambio

46 62 97 0 30

En la siguiente iteración tomamos el 97 y se ve que, no debe recorrerse hacia adelante por lo que no se hace ningún intercambio. Luego se realiza lo

mismo con el siguiente elemento recorriendo todo el vector e insertando el elemento en el lugar apropiado, obteniendo

0 46 62 97 30

esto se repite hasta obtener el vector ordenado.

Realizando el análisis del algoritmo se ve que el tiempo de ejecución es proporcional a

$$\sum_{i=1}^{n-1}\sum_{j=i}^{j=1} = \sum_{i=1}^{n-1} i = n\frac{n-1}{2}$$

La ventaja de este algoritmo es que es más simple que, el anterior y aún cuando su tiempo de proceso es también proporcional a n^2 es más eficiente y puede afinarse para ser más eficiente. El código siguiente muestra el algoritmo mejorado para ser más eficiente. El tiempo de proceso sigue siendo proporcional a $O(n^2)$

```
void Insercion2(int[] x) {
    int i, j, temp;
    int n = x.length;
    for (i = 1; i < n; i++) {
    //los datos estanordenados hasta i
      temp=x[i];
      for (j = i; j > 0 && x[j - 1] > temp; j--) {
        x[j] = x[j-1];
      }
      x[j] = temp;
    // con esta asignacion restablecemos la invariante
    }
  }
```

Como se observa se ha mejorado la parte que corresponde a constuir el espacio para introducir el elemento siguiente reduciendo el número de intercambios.

Ordenación por selección

En el algoritmo de inserción la invariante indica que los valores posteriores al indice i pueden ser mayores o menores que el último valor ordenado. Podemos cambiar esta invariante haciendo que el último valor ordenado sea

menor que todos los elementos por ordenar. Esto se representa graficamente como sigue:

Para implementar el concepto expresado en la invariante se escoge el elemento más pequeño y se reemplaza por el primer elemento, luego repetir el proceso para el segundo, tercero y así sucesivamente. El resultado es el siguiente programa

```
void Seleccion(int[] x) {
  int i, j;
  int min, temp;
  int n = x.length;
  for (i = 0; i < n - 1; i++) {
    min = i;
    for (j = i + 1; j < n; j++) {
      if (x[j] < x[min])
        min = j;
    }
    temp = x[i];
    x[i] = x[min];
    x[min] = temp;
  }
}
```

Como se observa este algoritmo es similar al algoritmo de inserción con tiempo similar y proporcional a $O(n^2)$

Algoritmo de clasificación rápida

El qsort cuyo nombre en inglés es QuickSort o también denominado algoritmo de clasificación rápida ya fue publicada por C.A.R. Hoare en 1962. y se basa en dividir el vector en dos mitades utilizando un elemento denominado pivote de tal forma que todos los elementos mayores al pivote estén en un vector y los restantes en el segundo vector. Como pivote inicial se puede tomar el primer elemento o uno al azar.

El siguiente gáfico muestra la invariante, vemos que para cada punto los valores de la derecha siempre son mayores.

Este proceso se repite en forma recursiva hasta tener el vector ordenado. A partir de este enfoque podemos realizar una primera aproximación a lo que viene ser el algoritmo:

```
Ordenar (l,u){
  if (u >= l) {
    hay como maximo un elemento
    por lo tanto no hacer nada
  }
  hallar un pivote p para dividir en vector en dos
  Ordenar (l,p-1)
  Ordenar (p+1,u)
}
```

Aquí como dijimos el problema es encontrar el punto de partición denominado pivote. Utilizando el concepto que la invariante es que: los valores entre x[m]

y x[i] son menores al pivote x[l] podemos construir el siguiente código para hallar el pivote.

```
int m = l;
//invariante los valores entre x[m] y x[i]
//son menores al pivote x[l]
for (int i = l + 1; i <= u; i++) {
      if (x[i] < x[l])
      {
        temp = x[++m];
        x[m] = x[i];
        x[i] = temp;
      }
    }
temp = x[l];
    x[l] = x[m];
    x[m] = temp;
```

Supongamos que tenemos los siguientes datos:

27 2 81 81 11 87 80 7

Iniciamos un contador m en el primer valor y tomando $x[l] = 27$ como pivote y se recorre el vector comparando cada uno de los valores con el pivote haciendo un intercambio entre (m+1) y la posición del vector, obteniendo en el primer intercambio entre el número 2 que es menor que el pivote y el número de la posición m que es casualmente el 2, obteniendo:

27 2 81 81 11 87 80 7

El segundo intercambio es entre el 11 que es menor que el pivote y el 81 obteniento:

27 2 11 81 81 87 80 7

El tercer intercambio se produce con el 7 y el 81 obteniendo

27 2 11 7 81 87 80 81

Para finalizar intercambiamos el pivote con la posición del último número menor al pivote, obteniendo un vector donde todos los elementos menores al pivote están a la izquierda y los mayores a la derecha.

7 2 11 27 81 87 80 81

Este proceso se puede repetir recursivamente para el vector a la izquierda del pivote y para el que está a la derecha obteniento finalmente:

```
  private void qsort(int[] x, int l, int u) {
    if (l >= u)
      return;
    int m = l, temp;
//invariante los valores entre x[m] y x[i]
// son menores al pivote x[l]
    for (int i = l + 1; i <= u; i++) {
      if (x[i] < x[l])
      {
        temp = x[++m];
        x[m] = x[i];
        x[i] = temp;
      }
    }
    temp = x[l];
    x[l] = x[m];
    x[m] = temp;
    qsort(x, l, m - 1);
    qsort(x, m + 1, u);
  }
```

Analizando el código anterior podemos ver que el tiempo de proceso se puede calcular, utilizando los métodos vistos en los capítulos anteriores

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 2T(n/2) + n & \textit{en otros casos.} \end{cases}$$

En la ecuación anterior n corresponde al ciclo for que recorre todo los elementos del vector reordenando el vector según el pivote. Las siguientes dos llamadas al programa hacen que $2T(n/2)$. Aplicando el teorema master obtenemos que el tiempo de proceso es proporcional a $n \log(n)$.

Aunque no se divide exactamente a la mitad se ha tomado este valor para mostrar como mejora la eficiencia . En la realidad se puede mejorar en función de como está el conjunto de datos y como se elige al pivote.

En el caso de que el vector esté previamente ordenado el tiempo de proceso será mayor. El proceso de elegir un pivote aleatoriamente produce un

algoritmo más estable. Es posible optimizar el método de ordenación rápida pero no se tratará en el texto. Cuando uno requiere un algoritmo de estas características normalmente se recurre a las rutinas ya implementadas en las librerías del lenguaje.

Se pueden mejorar aún más los algoritmos de clasificación? En casos particulares se puede obtener algoritmos proporcionales a $O(n)$ en el caso más general, el mejor algoritmo es proporcional a $n\log(n)$. Una demostración se puede leer en el texto de [GB96]

Algoritmos lineales

Considere las siguientes condiciones para los datos de entrada, se dispone de memoria, suficiente, los datos son enteros positivos sin repetición en el rango de 0 a N, donde N es el número máximo existente. Para ordenar estos números se puede utilizar cualquiera de los algoritmos descritos para ordenar.

Para mejorar el proceso de ordenar consideremos la siguiente estructura

```
BitSet a = new BitSet(max + 1);
```

Si cada posición del vector representa a uno de los números del vector se utiliza un bit para almacenar cada uno de los números. En este caso es posible leer los números de entrada y encender el bit que le corresponde. Recorriendo el vector se pueden imprimir los números a los que corresponden los bits encendidos en forma ordenada.

El resultado proporciona el siguiente algoritmo.

```
public void bitSort(int[] x) {
  int max=0, j=0;
  for(int i =0; i< x.length; i++){
    if (x[i] > max)
      max=x[i];
  }
  BitSet a = new BitSet(max + 1);
  // Hasta aqui todos los valores de
  // a estan en cero
  for(int i =0;i< x.length; i++)
    a.set(x[i]);
  for(int i =0;i<= max; i++){
    if (a.get(i))
```

```
        x[j++]=i  ;
    }
  }
```

Analizando el algoritmo se ve que se tienen tres ciclos que recorren todos los datos dando un tiempo de ejecución proporcional a $O(n)$.

Como se ve en este método antes de aplicar el algoritmo ya conocemos la posición que cada elemento debe tomar. En estos casos se puede desarrollar algoritmos de tiempo lineal.

Otro ejemplo de clasificación en tiempo lineal es el problema de ordenar la bandera de Bolivia. Supongamos que tenemos una bandera que tiene los colores desordenados (precondición), tal como muestra el gráfico:

Una vez ordenado (postcondición) el resultado a obtener sería

Para ordenar utilizaremos la siguiente invariante:

Para resolver este problema ya conocemos el lugar exacto de cada uno de los colores de la bandera. Rojos a la izquierda, amarillos al centro, y verdes a la derecha. El proceso que se sigue, manteniendo la invariante que nos dice que desde el último amarillo hasta el primer verde no está ordenado.

Llevamos sucesivamente los rojos a la izquierda y los verdes a la derecha. Lo amarillos quedan al centro por lo que queda ordenado. El programa java ejemplifica el proceso:

```
/**
 * Programa para ordenar la bandera de Bolivia
 *
```

```
 * @author Jorge Teran
 *
 */
public class Bandera {
    public static final void main(String[] args)
            throws Exception {
        char[] bandera = { 'R', 'A', 'A', 'V',
                'A', 'V', 'V', 'R', 'A', 'R' };
        for (int i = 0; i < 10; i++)
            System.out.print(bandera[i] + " ");
        System.out.println("");
        int r = 0, a = 0, v = 10;
        char t;
        while (a != v) {
            if (bandera[a] == 'R') {
                t = bandera[r];
                bandera[r] = bandera[a];
                bandera[a] = t;
                r++;
                a++;
            } else if (bandera[a] == 'A') {
                a++;
            } else if (bandera[a] == 'V') {
                t = bandera[a];
                bandera[a] = bandera[v - 1];
                bandera[v - 1] = t;
                v--;
            }
        }

        for (int i = 0; i < 10; i++)
            System.out.print(bandera[i] + " ");
        System.out.println("");
    }
}
```

Está probado que cualquier algoritmo basado en comparaciones toma un tiempo $O(n \log n)$. Como es posible que ordenar la bandera tome un tiem-

po lineal? La respuesta es que en el proceso conocemos la posición de los elementos de la bandera. ,

Comparación práctica de la eficiencia de los algoritmos

A fin de comparar los algoritmos de clasificación es necesario hacer una corrida de los mismos sobre el mismo conjunto de datos. Para esto realizamos el siguiente programa:

```
/**Programa para probar los diferentes
 * algoritmos de ordenación
  * @author Jorge Teran
 *
 */
import java.util.*;

public class pruebaMetodos {

  public static int MAXIMO = 10000;

  public static void main(String[] args) {
    //generar MAXIMO  números sin repetidos
    int[] x = new int[MAXIMO];
    int[] y = new int[MAXIMO];
    for (int i = 0; i < MAXIMO; i++)
      x[i]=i;
    Random gen = new Random();
    int temp, j;
    for (int i = 0; i < MAXIMO; i++){
      j = gen.nextInt(MAXIMO);
      temp = x[i];
      x[i] = x[j];
      x[j]=temp;
    }

    System.out.println("Tamaño de la instancia"+MAXIMO);
        long tiempoInicio, tiempoFin, tiempoTotal;
    Metodos sort =new Metodos();
```

```
    //obtener una copia de x para ordenar
//para siempre utilizar el mismo conjunto de datos
System.arraycopy(x,0,y,0,MAXIMO);
tiempoInicio = System.currentTimeMillis();
sort.Burbuja(y);
tiempoFin = System.currentTimeMillis();
tiempoTotal = tiempoFin - tiempoInicio;
System.out.println(" Método Burbuja "
     + " Tiempo de proceso " + tiempoTotal);

System.arraycopy(x,0,y,0,MAXIMO);
tiempoInicio = System.currentTimeMillis();
sort.Insercion(y);
tiempoFin = System.currentTimeMillis();
tiempoTotal = tiempoFin - tiempoInicio;
System.out.println(" Método Insercion "
     + " Tiempo de proceso " + tiempoTotal);

System.arraycopy(x,0,y,0,MAXIMO);
tiempoInicio = System.currentTimeMillis();
sort.Insercion2(y);
tiempoFin = System.currentTimeMillis();
tiempoTotal = tiempoFin - tiempoInicio;
System.out.println(" Método Insercion2 "
     + " Tiempo de proceso " + tiempoTotal);

System.arraycopy(x,0,y,0,MAXIMO);
tiempoInicio = System.currentTimeMillis();
sort.Seleccion(y);
tiempoFin = System.currentTimeMillis();
tiempoTotal = tiempoFin - tiempoInicio;
System.out.println(" Método Seleccion "
     + " Tiempo de proceso " + tiempoTotal);

System.arraycopy(x,0,y,0,MAXIMO);
tiempoInicio = System.currentTimeMillis();
sort.quickSort(y);
tiempoFin = System.currentTimeMillis();
```

```
    tiempoTotal = tiempoFin - tiempoInicio;
    System.out.println(" Método quickSort "
         + " Tiempo de proceso " + tiempoTotal);

    System.arraycopy(x,0,y,0,MAXIMO);
    tiempoInicio = System.currentTimeMillis();
    sort.bitSort(y);
    tiempoFin = System.currentTimeMillis();
    tiempoTotal = tiempoFin - tiempoInicio;
    System.out.println(" Método bitSort "
         + " Tiempo de proceso " + tiempoTotal);
  }
}
```

obteniendo los siguientes resultados para el proceso de 100.000 números generados aleatoriamente.

Método	Tiempo de proceso en miliseg
Burbuja	265.251
Inserción	114.364
Inserción2	86.795
Selección	155.854
quickSort	110
bitSort	30

Hay que observar que antes de cada proceso hemos realizado una copia del vector original para que la comparación sea adecuada al mismo conjunto de datos. En función del tamaño del conjunto el comportamiento de los algoritmos será diferente y es posible comprobarlo experimentalmente. El tiempo que demoró el método de Java para 100.000 números aleatorios 60 milisegundos.

5.3.3. Laboratorio

1. Probar experimentalmente para que valores de n el sort de burbuja toma un tiempo menor al qsort. Solo se requiere hallar un n que cumpla la relación

$$n\frac{n-1}{2} < n\log(n)$$

Algebraicamente podemos hallar éstos valores. Pero sabemos que una implementación particular está afectada por una constante que varía según el grado de eficiencia con el que fue programado, el lenguaje o las características del conjunto de datos.

Lo que, se pide es que realice una implementación de los algoritmos explicados y encuentre experimentalmente el número de elementos para los que un algoritmo sea mejor que otro.

Este proceso debe realizarse tanto para un vector ordenado de manera aleatoria, ascendente y descendente.

2. Hallar el tiempo en miliseg. de ordenar una instancia y llenar el cuadro siguiente para un vector ordenado de manera aleatorea, ascendente y descendente.

Tamaño	10	10^2	10^3	10^4	10^5	10^6
Burbuja						
Inserción						
Inserción2						
Selección						
Quicksort						
Del Java						
Bitsort						

3. Hallar las constantes experimentalmente

Burbuja	$aO(n^2)$	$a =$
Inserción	$aO(n^2)$	$a =$
Inserción2	$aO(n^2)$	$a =$
Selección	$aO(n^2)$	$a =$
Quicksort	$aO(n \log n)$	$a =$
Del Java	$aO(n \log n)$	$a =$
Bitsort	$aO(n)$	$a =$

5.3.4. Ejemplo de aplicación

5.3.5. Ejemplo 1

Se presenta el problema 120 publicado en el Juez de Valladolid España y que es un típico problema de ordenación.

Cocinar panqueques en una sartén es bastante complicado porque sin importar cuanto se esmere todos tendrán un diámetro diferente. Para la presentación de la pila usted los puede ordenar por tamaño de tal manera que cada panqueque es más pequeño que los que están más abajo. El tamaño está dado por el diámetro.

La pila se ordena por una secuencia de volteos. Un volteo consiste en insertar una espátula entre dos panqueques y volteando todos los panqueques en la espátula (colocando en orden reverso la subpila). Un volteo se especifica indicando la posición del panqueque en la base de la subpila a ser volteado. El panqueque de la base tiene la posición 1 y el de encima n.

La pila se especifica dando el diámetro de cada panqueque en orden en que aparecen. Por ejemplo considere las tres pilas de panqueques en el cual 8 es el panqueque encima de la pila de la izquierda

8	7	2
4	6	5
6	4	8
7	8	4
5	5	6
2	2	7

La pila de la izquierda puede ser transformada a la pila del medio por *volteo*(3). La del medio puede transformarse en la de la derecha con *volteo*(1).

Entrada La entrada consiste en secuencias de pilas de panqueques. Cada pila consiste de 1 a 20 panqueques y cada panqueque tendrá un diámetro entero entre 1 y 100, la entrada se termina con un fin de archivo. Cada pila está dada en una fila con el panqueque de arriba primero y el de la base al final. Todos ellos separados por un espacio.

Salida Para cada pila de panqueques su programa debe imprimir la pila original en una línea seguido de una secuencia de volteos que ordena la pila de panqueques de tal forma que el panqueque más grande este abajo y el más pequeño arriba, la secuencia de volteos debe terminar con un 0 indicando que no se requieren más volteos. Cuando la pila está ordenada no se deben realizar más volteos.

Ejemplo entrada

```
1 2 3 4 5
5 4 3 2 1
5 1 2 3 4
```

Ejemplo Salida

```
1 2 3 4 5
0
5 4 3 2 1
1 0
5 1 2 3 4
1 2 0
```

Solución Para resolver el ejercicio en cuestión lo primero que se debe evaluar es si se parece a algún algoritmo, conocido de clasificación. El segundo paso es el de determinar si el algoritmo provisto por el lenguaje de programación resuelve el problema, pues no vamos a contruir un algoritmo existiendo uno que resuelve el problema.

Como se ve es muy parecido al método de inserción por lo que codificaremos éste y agregaremos los requerimientos del problema.

Primero un método para invertir la pila de panqueques que denominaremos reverso.

```
  void reverso(int[] pan, int l) {
    int temporal = 0;
    int i = 0;
    int j = l;
    while (i < j) {
      temporal = pan[i];
      pan[i] = pan[j];
      pan[j] = temporal;
      i++;
      j--;
    }
    return;
```

En segundo lugar codificamos el método buscando el lugar apropiado para realizar la inserción que se consigue llamando a reverso

```
  void inicio() {
...
...
  c = 0;
      l = l - 1;
      ver(pan, l); // listar el orden original
```

```
      for (int i = l; i > 0; i--) {
        max = i;
        for (int j = i - 1; j >= 0; j--)
          if (pan[j] > pan[max])
            max = j;
        if (max != i) {
          if (max == 0)
            System.out.print((l + 1 - i) + " ");
          else
            System.out.print((l+1-max) + " " + (l+1-i)
                + " ");
          reverso(pan, max);
          reverso(pan, i);
        }
      }
    System.out.println("0");
    }
  }

  void ver(int[] pan, int l) {
//lista los datos instroducidos
    for (int i = 0; i < l; i++)
      System.out.print(pan[i] + " ");
    System.out.print(pan[l] + "\n");
    }
  }
```

Intencionalmente hemos omitido la parte de la lectura de datos porque es irrelevante en la explicación del procedimiento de resolución.

5.3.6. Ejemplo 2

Otro ejemplo típico es el ejercicio 10815, que se refiere a construir un diccionario a partir de un texto.

Andy, tiene un sueño - quiere producir su propio diccionario. Esto no es una tarea fácil para él, como el número de palabras que él sabe no son suficientes. En lugar de pensar todas las palabras por sí mismo, tiene una idea brillante. De su biblioteca recoge uno de sus libros de cuentos favoritos,

de la que se copia todas las palabras distintas. Al organizar las palabras en orden alfabético se construye automáticamente. Por supuesto, es una tarea que consume tiempo, y aquí es donde un programa de ordenador es muy útil.

Se le pedirá que escriba un programa que enumere todas las palabras en el texto de entrada. En este problema, una palabra se define como una secuencia consecutiva de caracteres, en mayúsculas y / o minúsculas. Palabras con una sola letra, también deben considerarse. Además, el programa debe considerar igual las mayúsculas y minúsculas. Por ejemplo, palabras como "Manzana", "manzana.º "MANZANA"se debe considerarse como la misma.

Entrada El archivo de entrada es un texto de no más de 5000 líneas. Una línea de entrada tiene como máximo 200 caracteres. La entrada termina con EOF.

Salida La salida debería dar la lista de palabras que aparecen en el texto de entrada, cada palabra en una línea. Todas las palabras deben estar en minúsculas, en orden alfabético. Usted puede estar seguro de que el número de palabras diferentes en el texto no excede de 5000.

Ejemplo entrada

```
Adventures in Disneyland

Two blondes were going to Disneyland when they
came to a fork in the
road. The sign read: "Disneyland Left."

So they went home.
```

Ejemplo Salida

```
a
adventures
blondes
came
disneyland
fork
going
home
in
left
```

```
read
road
sign
so
the
they
to
two
went
were
when
```

Solución Las condiciones especifican que a lo sumo habrán 5000 líneas de 200 caracteres. Indica que no se debe hacer diferencia entre minúsculas y mayúsculas y además se deben eliminar los signos de puntuación. Para esto tomaremos en cuenta algunos aspectos:

- En la lectura utilizando la clase *Scanner* con la opción *.next()* se lee la próxima palabra.
- Para eliminar los caracteres de puntuación utilizamos una expresión regular que nos permite identificar los caracteres, y luego los reemplazamos por *null*. La expresión regular es *[::,;]*.
- Para generar el resultado pedido primero ordenamos los datos con las instrucción *Arrays.sort*. Esto con la finalidad de poder imprimir los resultados eliminando las palabras repetidas.

El código resultante es el siguiente:

```
/*
 * Solucion al ejemplo 10815
 *
 * @author Jorge Teran
 */
import java.util.Scanner;
import java.util.Arrays;

public class P10815 {
```

```
    public static void main(String[] args) {

        Scanner entrada = new Scanner(System.in);
        String[] palabras = new String[10000000];
        int j, i = 0;
        while (entrada.hasNext()) {
            palabras[i] = entrada.next()
                    .replaceAll("[\".:,;]", "")
                    .toLowerCase();
            i++;
        }

        Arrays.sort(palabras, 0, i);
        String anterior = palabras[0];
        System.out.println(palabras[0]);
        for (j = 1; j < i; j++) {
            if (palabras[j].compareTo(anterior) > 0) {
                System.out.println(palabras[j]);
                anterior = palabras[j];
            }
        }
    }
}
```

5.4. Ejercicios

Resuelva los siguientes ejercicios del Juez de Valladolid:

123, 152, 156, 195, 263, 264, 299, 340, 400, 450, 482, 555, 612, 642, 755, 843, 880, 10008, 10041, 10050, 10062, 10098, 10107, 10125, 10132, 10138, 10194, 10202, 10277, 10327, 10420, 10474, 10619, 10698, 10745, 10763, 10785, 10810, 10815, 10881, 10905, 11224, 11242, 11321, 11373, 11386, 11389, 11411, 11413, 11462, 11495, 11516, 11579, 11627, 11646, 11649, 11751.

Capítulo 6

Combinatoria básica

6.1. Introducción

La combinatoria es la disciplina de las matemáticas que se refiere a las técnicas de contar. En realidad lo que trata de responder es a la pregunta cuántos son? sin tener que contarlos.

En general las técnicas de contar proveen una serie de alternativas al momento de tener que escribir programas complicados que cuenten todos los valores y permiten tener expresiones, que solucionen el problema aplicando fórmulas simples.

6.2. Técnicas básicas para contar

En esta parte recordaremos una serie de fórmulas de la combinatoria y mostraremos como son aplicadas en la programación. No desarrollaremos los conceptos matemáticos porque no corresponde a lo que el texto pretende, que es mostrar la forma de resolver problemas.

Regla del Producto Si un conjunto A tiene $|A|$ posibilidades y el B tiene $|B|$ posibilidades entonces existen $|A|\,x\,|B|$ formas de combinar éstos. Por ejemplo si tengo 4 tipos de automóviles y 3 colores se tienen 12 combinaciones. Estas combinaciones pueden representarse en una matriz cuadrada almacenando en cada celda una combinación. En el caso de tres conjuntos con una matriz de tres dimensiones, vea que el espacio ocupado o el recorrido de los elementos se hace cada vez más grande

Regla de la suma Si un conjunto A tiene $|A|$ posibilidades y el B tiene $|B|$ posibilidades entonces existen $|A| + |B|$ posibles formas en las que A o B pueden ocurrir. En el ejemplo anterior si uno de los automóviles, o de los colores es errado, existen $4 + 3 = 7$ posibles ítems fallados.

Regla inclusión y exclusión Esta es una regla generalizada de la suma. Supongamos que tenemos automóviles de 5 colores y minibuses de 6 colores y queremos conocer cuántos colores diferentes existen. En este caso tenemos dos conjuntos que se superponen. Esto significa sumar la cantidad de colores de ambos y restar los que sean comunes a ambos que se expresa como:

$$|A| \cup |B| = |A| + |B| - |A| \cap |B|$$

Permutaciones Una permutación es un arreglo de n ítems donde cada ítem aparece solo una vez. Existen 3! permutaciones para un conjunto de tres elementos. Si los elementos son 123 las permutaciones vienen a ser 123 132 231 213 312 321. Para un n arbitrario son $n!$. hay que observar que a medida que aumenta n el número de permutaciones aumenta muy rápidamente.

Subconjuntos Un subconjunto es la selección de algunos elementos de n posibles ítems. Para un conjunto arbitrario existen 2^n posibles subconjuntos, por ejemplo para 3 elementos existen 2^3 subconjuntos, ésto es: 1,2,3,12,13,23,123 y el conjunto vacío. A medida que crece n, rápidamente se llega a tiempos de proceso muy grandes.

Cadenas Cuando tratamos con cadenas o sea una secuencia de ítems con repetición, se tienen m^n distintas secuencias. Para 3 elementos con repetición se tienen 27 casos que son 111, 112, 113, 121, 122, 123, 131, 132, 133, 211, 212, 213, 221, 222, 223, 231, 232, 233, 311, 312, 313, 321, 322, 323, 331, 332, 333. El número crece más rápidamente a medida que crece m.

En muchos casos podemos encontrar que lo que tenemos es una función *biyectiva* o sea un asociación uno a uno con otro conjunto. En este caso podemos resolver el problema de contar cualquiera de los dos conjuntos. Por ejemplo si decimos que cada planta está en una maceta da lo mismo contar las plantas o las macetas. Si una maceta pudiera tener más de una planta esto ya no es cierto.

6.3. Coeficientes binomiales

Los coeficientes binomiales se refieren a calcular

$$\left(\begin{array}{c} n \\ k \end{array} \right)$$

que es el número de formas posibles de escoger k elementos de n posibilidades. Que elementos se cuentan más comunmente?

Comités Cuantas maneras hay de formar un comité de k miembros con n personas. La respuesta se halla por la definición.

Caminos en un matriz Cúantas maneras hay de viajar desde la esquina superior hasta la esquina inferior derecha en una matriz de m x n caminando solamente hacia abajo o hacia la izquierda? Cada camino debe consistir de $m + n$ pasos n hacia abajo y m a la derecha, por lo que existen:

$$\left(\begin{array}{c} n+m \\ n \end{array} \right)$$

Coeficientes de $(a + b)^n$ Cuál es el coeficiente de $a^k b^{n-k}$ claramente $\left(\begin{array}{c} n \\ k \end{array} \right)$ porque cuenta el número de formas que podemos utilizar para escoger k elementos de n.

Triángulo de Pascal Para calcular los coeficientes binomiales de puede realizar utilizando directamente la fórmula

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$$

Sin embargo los cálculos intermedios pueden hacer que el trabajo con números grandes sea inadecuado o de tiempo muy largo, una alternativa es utilizar el triángulo de Pascal.

n	p(n)	p(n,1)	p(n,2)	p(n,3)	p(n,4)	p(n,5)	p(n,6)
1	1						
2	1	1					
3	1	2	1				
4	1	3	3	1			
5	1	4	6	4	1		
6	1	5	10	10	5	1	
7	1	6	15	20	15	6	1

Lo interesante de éste triángulo es que calcula los coeficientes binomiales sin tener que realizar factoriales. Para calcular n tomados de k en tiempo constante podemos construir el triángulo de Pascal

```
/**Programa para constuir el
  * triangulo de Pascal
  * @author Jorge Teran
  *
*/

  public static void triángulo(int[][] x) {
  // Invariante cada linea debe comenzar con uno
  for (int i=0;i<MAXIMO;i++)
      x[i][0]=1;
// Invariante cada linea debe terminar con uno
    for (int i=0;i<MAXIMO;i++)
      x[i][i]=1;
    for (int i=1;i<MAXIMO;i++)
// Invariante cada linea debe sumar 2**i
      for (int j=1;j<=i;j++)
        x[i][j]=x[i-1][j-1]+x[i-1][j];
    }
  }
```

luego es posible hallar n tomados de k

```
  public static int nTomadosDek(
     int [][] x,int n, int k) {
    return x[n][k];
  }
```

Ahora explicamos cuáles son las invariantes del triángulo de Pascal. Primero cada línea debe comenzar y terminar con 1. Y en segundo lugar en cada línea la suma de sus valores da 2^i. Esto es

$$\sum_{k=0}^{k=i} \left(\begin{array}{c} i \\ k \end{array} \right) = 2^i$$

Fíjese que en ésta implementación se utiliza solo la mitad del arreglo bidimensional creado.

Una propiedad interesante es que la suma de las diagonales del triángulo da los números de Fibonacci, que se presentan en la siguiente sección, veamos esto con los datos del ejemplo:

Diagonal	Suma
1	$1 = 1$
2	$1 = 1$
3	$1 + 1 = 2$
4	$2 + 1 = 3$
5	$1 + 3 + 1 = 5$
6	$3 + 4 + 1 = 8$

Otras propiedades del triángulo de Pascal pueden verse en http://mathworld.wolfram.com/

6.4. Algunas secuencias conocidas

6.4.1. Números de Fibonacci

Los números de Fibonacci son la secuencia de números definidos por la ecuación:

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

Teniendo definido que $T(0) = 0$ y $T(1) = 1$, se genera la siguiente secuencia 1, 1, 2, 3, 5, 8, 13, 21, ...

Los números Fibonacci aparecen en diferentes ámbitos, desde la criptografía, como menciona Dan Brown en su novela Código DaVinci, en la naturaleza existen ejemplos, en cristalografía y en la descripción de la posición de las semillas en las plantas de girasol.

Para hallar el término enésimo de la secuencia podemos utilizar los métodos de resolución de recurrencias obteniendo:

$$T(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2}^n)$$

Una explicación detallada se halla en el texto de Matemática Discreta de Grimaldi [Gri77].

Los números de Fibonacci satisfacen también la siguiente relación del máximo común divisor

$$MCD(F_n, F_m) = F_{MCD(n,m)}$$

También se puede ver que los números de Fibonacci tienen la característica de que al dividirse módulo m, producen una secuencia cíclica.

Fibonacci	mod 2	mod 3	mod 4
1	1	1	1
1	1	1	1
2	0	2	2
3	1	0	3
5	1	2	1
8	0	2	0
13	1	1	1
21	1 0	0	1
34	0 0	1	2
55	0 0	1	3

6.4.2. Números Catalanes

Los números Catalanes son los números que están asociados a la fórmula

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix}$$

Los primeros números Catalanes son: 1,1,2,5,14,42,132,429 y se pueden obtener como sigue:

número Catalán	Valor
C_0	1
C_1	1
C_2	$C_0C_1 + C_1C_0 = 2$
C_3	$C_0C_2 + C_1C_1 + C_2C_0 = 5$
C_4	$C_0C_3 + C_1C_2 + C_2C_1 + C_3C_0 = 14$

Estos números se presentan en diferentes problemas, veamos por ejemplo la siguiente secuencia $\{-1, -1, -1, 1, 1, 1\}$. ¿De cuántas maneras se pueden ordenar para que las sumas parciales de los elementos sea positiva?

Estos elementos son $\{1,1,1,-1,-1,-1\}$, $\{1,1,-1,1,-1,-1\}$, $\{1,1,-1,-1,1,-1\}$, $\{1,-1,1,1,-1,1\}$ y $\{1,-1,1,-1,1,-1\}$.

Esta secuencia corresponde al número Catalán 3 que es 5, que también podemos calcular de la combinatoria

$$\frac{1}{3+1}\left(\begin{array}{c}6\\3\end{array}\right)=5$$

Algunos otros ejemplos de aplicación de los números Catalanes son los siguientes.

- ¿De cuántas formas se pueden colocar 3 paréntesis de tal manera que cada paréntesis de apertura tenga su respectivo de cierre? La respuesta es el número Catalán 3, veamos ((())), ()((()), (())(), (()()) y ()()()

- Dado un polígono regular de n vértices, de cuántas formas se puede dividir en $n-2$ triángulos, si las diferentes orientaciones se cuentan en forma diferente?. Un cuadrado se puede dividir en dos triángulos con una línea entre sus vértices las posibilidades son 2. En un pentágono se hace con, dos líneas y se obtienen 3 triángulos y el número de casos es número Catalán 3 o sea 5.

- Dados dos candidatos A y B que en una elección, cada uno obtiene n votos (empate). De cuantas maneras se puede contar los votos de tal forma que al realizar el recuento, el candidato A siempre esté por delante del candidato B. Este también el número Catalán n.

6.4.3. Número Eulerianos

Sea $p=\{a_1,a_2,...a_n\}$, deseamos conocer todas las permutaciones que cumplen la relación $a_i<a_{i+1}$. Por ejemplo si $p=\{1,2,3,4\}$ se construyen todas las permutaciones de p y contamos los casos en los que se cumple la relación de orden:

Permutación	Cuántos cumplen	Permutación	Cuántos cumplen
1234	3	2134	2
1243	2	2143	1
1324	2	2314	2
1342	2	2341	2
1423	2	2413	2
1432	1	2431	1
3124	2	4123	2
3142	1	4132	1
3214	1	4213	1
3241	1	4231	1
3412	2	4312	1
3421	1	4321	0

por ejemplo la permutación 2341 tiene $2 < 3$ y $3 < 4$ que son dos casos que cumplen la relación de orden. Cuántas permutaciones de 4 elementos tienen dos casos que cumplen la relación de orden?. Este valor está representado por número Euleriano

$$\left\langle \begin{array}{c} n \\ k \end{array} \right\rangle = \sum_{j=0}^{k+1} (-1)^j \left(\begin{array}{c} n+1 \\ j \end{array} \right) (k-j+1)^n$$

si contamos los valores del ejemplo vemos que son 11.

Si organizamos estos números en un triángulo se obtiene el triángulo Euleriano

n	$\left\langle \begin{array}{c} n \\ 0 \end{array} \right\rangle$	$\left\langle \begin{array}{c} n \\ 1 \end{array} \right\rangle$	$\left\langle \begin{array}{c} n \\ 2 \end{array} \right\rangle$	$\left\langle \begin{array}{c} n \\ 3 \end{array} \right\rangle$	$\left\langle \begin{array}{c} n \\ 4 \end{array} \right\rangle$	$\left\langle \begin{array}{c} n \\ 5 \end{array} \right\rangle$
1	1					
2	1	1				
3	1	4	1			
4	1	11	11	1		
5	1	26	66	26	1	
6	1	57	302	302	57	1

Analizando la tabla de permutaciones realizada vemos que

$$\left\langle \begin{array}{c} 4 \\ 0 \end{array} \right\rangle = 1 \left\langle \begin{array}{c} 4 \\ 1 \end{array} \right\rangle = 11 \left\langle \begin{array}{c} 4 \\ 2 \end{array} \right\rangle = 11 \left\langle \begin{array}{c} 4 \\ 3 \end{array} \right\rangle = 1$$

corresponden al triángulo presentado.

Este triángulo puede generarse de la siguiente relación de recurrencia:

$$\left\langle \begin{array}{c} n \\ k \end{array} \right\rangle = k \left\langle \begin{array}{c} n-1 \\ k \end{array} \right\rangle + (n-k+1) \left\langle \begin{array}{c} n-1 \\ k-1 \end{array} \right\rangle \tag{6.4.1}$$

Una propiedad interesante de los números Eulerianos que, puede utilizarse como invariante es que:

$$\sum_{k=0}^{n} \left\langle \begin{array}{c} n \\ k \end{array} \right\rangle = n!$$

El presente programa genera dicho triángulo utilizando la recurrencia 6.4.1

```
public class Eulerian {
    /**
     * Programa para constuir el triangulo Euleriano
     *
     * @author Jorge Teran
     * @param args
     *            none
     */
    public final static int MAXIMO = 6;

    public static void main(String[] args) {
        int[][] x = new int[MAXIMO][MAXIMO];
        for (int i = 0; i < MAXIMO; i++)
            x[i][1] = 1;
        for (int i = 0; i < MAXIMO; i++)
            x[i][i] = 1;

        for (int i = 2; i < MAXIMO; i++)
            for (int j = 2; j <= i; j++)
                x[i][j] = (i - j + 1)
                        * x[i - 1][j - 1] + j
                        * x[i - 1][j];

        for (int i = 1; i < MAXIMO; i++) {
            System.out.println(" ");
```

```
            for (int j = 1; j <= i; j++)
                System.out.print(x[i][j] + " ");
        }

    }
}
```

6.4.4. Números de Stirling

Consideremos un conjunto con n elementos, el número total de subconjuntos que podemos formar es 2^n. Nos preguntamos cuántos conjuntos de k subconjuntos podemos formar que excluyan el elemento vacío y la unión de ellos, nos da el conjunto original?

Para comprender el problema considere el conjunto $p = \{1, 2, 3, 4\}$ todos los conjuntos de 2 elementos son:

1	{(1),(2,3,4)}
2	{(2),(1,3,4)}
3	{(3),(2,1,4)}
4	{(4),(2,3,1)}
5	{(1,2),(3,4)}
6	{(1,3),(2,4)}
7	{(1,4),(2,3)}

Como podemos calcular este valor?. Primeramente veamos que un conjunto de n elementos se puede dividir en dos conjuntos $\{A, \bar{A}\}$ y $\{\bar{A}, A\}$ Como tenemos dos conjuntos existen dos conjuntos vacíos, entonces el total de subconjuntos que puede tener A y $\bar{A}$ son $2^n - 2$. Como no nos importa el orden, vale decir que el complemento esté antes o después hay que tomar la mitad de los elementos, esto es:

$$\frac{2^n - 2}{2} = 2^n - 1$$

En el caso de dividir en dos subconjuntos es relativamente simple de hallar el resultado. En cambio cuando tenemos un número mayor se hace más complicado.

Estos números se denominan números de Stirling de segundo orden. Para calcular $S(n, k)$ se puede utilizar, al igual que los ejemplos anteriores la siguiente recurrencia:

$$S(n, k) = S(n - 1, k) + kS(n - 1, k)$$

Hay que tomar en cuenta que los límites son: $S(n,1) = 1$, y $S(n,n) = 1$.

6.4.5. Particiones enteras

Se quiere contar de cuántas formas se puede escribir un número entero positivo como la suma de enteros positivos. Ahora el orden de los sumandos es irrelevante. Cada una de estas formas se denomina partición y cada uno de los sumandos parte. Por ejemplo:

1	$5 = 1+1+1+1+1$
2	$5 = 1+1+1+2$
3	$5 = 1+2+2$
4	$5 = 1+1+3$
5	$5 = 2+3$
6	$5 = 1+4$
7	$5 = 5$

En este caso la partición de 5 tiene 7 posibles particiones. Este problema no lo desarrollaremos y mencionaremos la recurrencia

Así analizamos este ejemplo veremos que hay 1 partición de 5 elementos, 1 de 4, 2 de 3, 2 de 2 y 1 de 1. Sea $p(n,k)$ el número total de elementos de una partición de k elementos entonces el número de particiones

$$p(n) = \sum_{k=1}^{k=n} p(n,k)$$

Estos valores también nos proporcionan un triángulo que puede armarse con la siguiente recurrencia:

$$p(n,k) = p(n-1,k-1) + p(n-k,k)$$

Considerando que $p(1,1) = 1$ y que $p(n,k) =$ siendo $k > n$. El siguiente programa produce el resultado deseado.

```
public class ParticionEntera {
    /**
     * Programa para hallar las particiones enteras
     *
     * @author Jorge Teran
```

n	p(n)	p(n,1)	p(n,2)	p(n,3)	p(n,4)	p(n,5)	p(n,6)	p(n,7)
1	1	1						
2	2	1	1					
3	3	1	1	1				
4	5	1	2	1	1			
5	7	1	2	2	1	1		
6	11	1	3	3	2	1	1	
7	15	1	3	4	3	2	1	1

Cuadro 6.1: Triángulo para hallar las particiones

```
     * @param args
     *            none
     */
    public final static int MAXIMO = 8;

    public static void main(String[] args) {
        int[][] x = new int[MAXIMO][MAXIMO];
        for (int i = 1; i < MAXIMO; i++)
            for (int j = 1; j <= i; j++) {
                if (j == 1 || j == i)
                    x[i][j] = 1;
                else
                    x[i][j] = x[i - j][j]
                            + x[i - 1][j - 1];
            }
        for (int i = 1; i < MAXIMO; i++) {
            System.out.println(" ");
            for (int j = 1; j <= i; j++)
                System.out.print(x[i][j] + " ");
        }
    }
}
```

El resultado del proceso produce el triángulo que mostramos en el cuadro 6.1 donde la p(n,k) esta representado por la fila y la columna. Para hallar el número total de particiones solo hay que sumar los elementos de una fila. Estos están sumarizados en la columna p(n).

6.4.6. Ejemplo de aplicación

El Problema 369 - Combinaciones del juez de Valladolid nos presenta el siguiente enunciado:

Calcular el número de formas que N objetos tomados de M, en un tiempo dado pueden ser un gran desafío cuando N y/o M llegan a ser muy grandes. Por consiguiente hay que hacer un programa que calcule los siguientes: datos: $5 \leq n \leq 100$ y $5 \leq m \leq 100$ y $m \leq n$

Calcule el valor exacto de:

$$c = \frac{n!}{(n-m)!m!}$$

Se puede asumir que el valor final de C no sobrepasa un entero de 32 bits.

Como antecendente, el valor exacto de 100! es:

```
93,326,215,443,944,152,681,699,238,856,266,700,490,715,
968,264,381,621,468,592,963,895,217,599,993,229,915,608,
941,463,976,156,518,286,253,697,920,827,223,758,251,185,
210,916,864,000,000,000,000,000,000,000,000
```

Entrada y salida La entrada de este programa puede ser una o más líneas que contienen 0 ó más espacios, un valor para N, uno ó más espacios y un valor para M . La última línea de la entrada dos ceros. El programa debe terminar cuando lee esta línea.

La salida para el programa debe estar en la forma:

```
N things taken M at a time is C exactly.
```

Ejemplo de entrada

```
100  6
 20  5
 18  6
  0  0
```

Ejemplo de salida

```
100 things taken 6 at a time is 1192052400 exactly.
20 things taken 5 at a time is 15504 exactly.
18 things taken 6 at a time is 18564 exactly.
```

Para resolver el problema debemos acordarnos que no es posible realizar el cálculo de los factoriales porque el resultado no es posible almacenar en una variable entera. También podemos recordar que estos valores se pueden obtener del triángulo de Pascal. Al construir el triangulo solo se realizan sumas y analizando los límites solo es necesario definir una matriz cuadrada que cabe en la memoria sin problemas. Los valores que no es posible almacenar en una variable entera quedaran desbordados como números negativos. Como el enunciado especifica que la respuesta es menor a 2^31, los valores que generan éste tipo de resultado no se presentaran en los datos de entrada. Con estas salvedades podemos plantear el siguiente programa:

```
import java.util.Scanner;
/**
 * @author Jorge Teran
 *
 */
public class P369 {

 public static void main(String[] args) {

     int MAXIMO = 101;
     int[][] triangulo = new int[MAXIMO][MAXIMO];
     for (int i = 0; i < MAXIMO; i++)
         triangulo[i][0] = 1;
     for (int i = 0; i < MAXIMO; i++)
         triangulo[i][i] = 1;
     for (int i = 1; i < MAXIMO; i++)
         for (int j = 1; j <= i; j++)
             triangulo[i][j] = triangulo[i - 1][j - 1]
                + triangulo[i - 1][j];
     int m, n;
     Scanner entrada = new Scanner(System.in);
     while (entrada.hasNext()) {
         m = entrada.nextInt();
         n = entrada.nextInt();
         if ((m == 0) && (n == 0))
             break;
         System.out.println(m + " things taken " + n
```

```
                + " at a time is "
                + triangulo[m][n] + " exactly.");
    }
 }
}
```

Otra solución alternativa es basada en algunas propiedades del máximo común divisor [Rol01]. Los coeficientes binomiales cumplen la siguiente propiedad:

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{\left(\begin{array}{c} n-1 \\ k-1 \end{array} \right)}{q} \frac{n}{d}$$

Donde $d = mcd(n, k)$ y $q = k/d$. La demostración es muy simple, reemplazamos el valor de q, simplificamos;

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{\left(\begin{array}{c} n-1 \\ k-1 \end{array} \right)}{\frac{k}{d}} \frac{n}{d}$$

Obteniendo:

$$\left(\begin{array}{c} n-1 \\ k-1 \end{array} \right) \frac{n}{k} = \frac{(n-1)!}{(k-1)!(n-k)!} \frac{n}{k} = \left(\begin{array}{c} n \\ k \end{array} \right)$$

Esta solución nos permite realizar muchas simplificaciones que nos permiten obtener el mismo resultado. Inicializamos un vector con los valores de n,$n-1$... 1

```
for (i = 0; i < k; i++)
           a[i] = n - i;
```

Seguidamente podemos calcular el $d = gcd(a[i], q)$

```
for (j = 2; j <= k; j++)
    for (i = 0, q = j; d > 1; i++) {
         d = gcd(a[i], q);
         a[i] /= d;
         q /= d;
 }
```

El segundo for es para realizar todas las simplificaciones posibles a fin de reducir el tamaño de los resultados. Y finalmente multiplicamos los valores intermedios para obtener el resultado.

```
for (i = 0, resultado = 1; i < k; res *= a[i++]);
```

6.5. Ejercicios

Resuelva los siguientes ejercicios del Juez de Valladolid:

135, 146, 153, 326, 369, 495, 763, 900, 948, 10007, 10165, 10183, 10213, 10219, 10220, 10223, 10229, 10236, 10303, 10312, 10328, 10334, 10375, 10518, 10579, 10601, 10623, 10634, 10689, 10696, 10776, 10784, 10790, 10918, 11000, 11028, 11118, 11161, 11255, 11270, 11282, 11330, 11481, 11507, 11525, 11582, 11699.

Capítulo 7

Estructuras de datos elementales

7.1. Introducción

En el presente capítulo se hace una introducción a las estructuras de datos básicas. La intención de éste capítulo es orientar en el uso de estructuras que vienen implementadas en el lenguaje de programación Java. No se trata de introducir los conceptos que corresponden a los cursos de estructura de datos, sino más bien, de reutilizar las herramientas provistas.

7.2. Vectores

Las estructuras que proporciona el lenguaje Java para el manejo de vectores son dos: vectores estáticos, y vectores dinámicos. Los vectores estáticos permiten una ejecución más rápida del código y son de tamaño fijo en cambio los vectores dinámicos permiten cambiar su tamaño a cambio de un tiempo procesamiento adicional.

7.2.1. Vectores estáticos

Los vectores estáticos son los que normalmente utilizamos y se pueden definir como sigue:

```
objeto[] x =new objeto[tamaño];
```

Por ejemplo para definir un vector de enteros con capacidad de 10 números enteros la definición es como sigue:

```
int[] x =new int[10];
```

Para definir un vector de constantes se escribe como sigue:

```
private static final String[] colores =
        { "ROJO", "VERDE", "AZUL", "CAFE", "GRIS" };
```

El método disponible es *length* que permite conocer la longitud del vector. En el ejemplo $colores.length = 5$ y $x.length = 10$.

7.2.2. Vectores dinámicos

Los vectores dinámicos son dos: *ArrayList* y *Vector*. Estas dos estructuras son similares con la diferencia que *Vector* cada vez que se incrementa de tamaño duplica su capacidad. Comienza en 10 luego duplica a 20, después a 40 y así sucesivamente. Veamos un ejemplo

```
/**
 * Programa paran mostrar el funcionamiento de
 * la clase vector.
 * @author Jorge Teran
 * @param args
 *            none
 */public class VectorDinamico {

    {
        private static final String letras[] =
              { "a", "b", "c", "d","e", "f" };

        public static void main(String args[]) {
            Vector<String> vector = new Vector<String>();
            System.out.println(vector);
            System.out.printf("\nTamaño: %d\nCapacidad: %d\n",
                     vector.size(), vector.capacity());
            // añadir elementos
            for (String s : letras)
                vector.add(s);
```

```
            System.out.println(vector);

            System.out.printf("\nTamaño: %d\nCapacidad: %d\n",
                     vector.size(), vector.capacity());

            // añadir más elementos
            for (String s : letras)
                vector.add(s);
            System.out.println(vector);
            System.out.printf("\nTamaño: %d\nCapacidad: %d\n",
                     vector.size(), vector.capacity());
        }
    }
```

El resultado de la ejecución es como sigue:

```
[]

Tamaño: 0
Capacidad: 10
[a, b, c, d, e, f]

Tamaño: 6
Capacidad: 10
[a, b, c, d, e, f, a, b, c, d, e, f]

Tamaño: 12
Capacidad: 20
```

Antes de insertar valores el vector está vacío. Se insertan 6 valores y tiene una capacidad de 10. Cuando se inserta 6 valores adicionales el tamaño crece a 12 pero la capacidad se ha duplicado a 20.

Los métodos de Java para la colección *Vector* se describen en el cuadro

Método	Descripción
add(E elemento)	Agrega un elemento al vector.
clear()	Borra la lista.
capacity()	Muestra el espacio reservado para el vector.
addAll(int i,coleccion)	Agrega toda una colección.
addAll(int i,coleccion)	Agrega toda una colección en la posición i.
remove(int i)	Quita el elemento de la posición i.
set(int i, E elemento)	Cambia el elemento de la posición i.
isEmpty()	Devuelve true si el vector esta vacío.
contains(E elemento)	Devuelve true si el vector contiene el elemento.
size()	Devuelve el número de elementos.
firstElement()	Devuelve el primer elemento.
lastElement()	Devuelve el último elemento.

Los *ArrayList* administran la memoria como una lista y crecen dinámicamente. La capacidad es igual al tamaño. Veamos un ejemplo de como puede crecer un *ArrayList* de tamaño.

```
import java.util.ArrayList;
public class Crece {
  public static void main(String args[]) {
    ArrayList v = new ArrayList(5);
    for (int i = 0; i < 10; i++) {
      v.add(i);
    }
    System.out.println(v);
  }
}
```

En este ejemplo hemos definido una vector del tipo *ArrayList* con tamaño 5. Porteriormente se insertan 10 valores. Cuando se llega al límite de elementos el vector cambia dinámicamente de tamaño incrementando el mismo en uno. Cabe notar que no es posible acceder a valores que exceden el tamaño del mismo.

Los métodos de Java para la colección *ArrayList* se describen en el cuadro

Método	Descripción
add(E elemento)	Agrega un elemento al vector.
clear()	Borra la lista.
addAll(int i,coleccion)	Agrega toda una colección.
addAll(int i,coleccion)	Agrega toda una colección en la posición i.
remove(int i)	Quita el elemento de la posición i.
set(int i, E elemento)	Cambia el elemento de la posición i.
isEmpty()	Devuelve true si el vector esta vacío.
contains(E elemento)	Devuelve true si el vector contiene el elemento.
size()	Devuelve el número de elementos.

Como ejemplo vamos a crear una lista con 5 números enteros, hacer crecer la lista, modificar y listar los elementos de la misma.

```
import java.util.ArrayList;
/**
 * Programa para Mostrar el funcionamiento de
 * un ArrayList
 * @author Jorge Teran
 * @param args
 *            none
 */
public class EjemploArrayList {
    public static void main(String args[]) {
        ArrayList v = new ArrayList(5);
        for (int i = 0; i < 10; i++) {
            v.add(i);
        }
        // imprimir los elementos
        System.out.println(v);
        // imprimir el elemento 5
        System.out.println(v.get(5));
        // cambiar el elmento 3 por 100
        v.set(3, 100);
        // Eliminar el elmento 5
```

```
        v.remove(5);
        // imprimir el tamaño
        System.out.println(v.size());
        // recorrer la lista
        for (int i = 0; i < v.size(); i++) {
            System.out.print("Elemento " + i
                    + "=" + v.get(i) + " ");
        }

    }
}
```

7.3. Pilas

Una de las estructuras más simples es la pila. La estructura de pila es una lista en la que solo se pueden ingresar y sacar valores por un solo lugar. Estas pilas se denominan también listas LIFO (last in first out). Esto significa que el último valor introducido es el primero en salir.

En esta estructura solo se pueden insertar objetos al final, extraer el último objeto o mirar el último. En Java ésto equivale a los métodos push, pop y peek.

Para mostrar como se realiza esta implementación utilizaremos la clase Alumno, que almacena nombre y código.

```
public class Alumno {
    String nombre;
    int código;
public Alumno(String  nombre, int código) {
  this.nombre=nombre;
  this.código=código;
  }
public String verNombre() {
  return nombre;
  }
public int verCodigo() {
  return código;
  }
}
```

El siguiente código implementa la clase pila, ingresando objetos de tipo Alumno y luego listándolos. El siguiente programa realiza lo indicado.

```
import java.io.*;
import java.util.*;
/**
 * Ejemplo de manejo de pilas
 * @author Jorge Teran
 * @param args
 *            none
 */

public class pila {
  public static void main(String[] args) {
    // definir una pila de alumnos
    Stack<Alumno> pila = new Stack();
    // almacenar los alumnos
    Alumno alm1 = new Alumno("Juan", 11);
    pila.push(alm1);
    alm1 = new Alumno("Maria", 17);
    pila.push(alm1);
    alm1 = new Alumno("Jose", 25);
    pila.push(alm1);
    // Recuperar el elemento de encima
    System.out.println(pila.peek().verNombre());
    // Imprimir la pila
    while (!pila.isEmpty())
      System.out.println(pila.pop().verNombre());
  }
}
```

Para ejemplificar la utilización de pilas supongamos que queremos verificar si una secuencia de paréntesis y corchetes está bien anidada. Consideremos algunos casos

- La secuencia (([])) es una secuencia bien anidada.
- La secuencia (([)]) es una secuencia mal anidada, un corchete no puede cerrarse con un paréntesis.

- La secuencia ((()) está mal, porque le falta un paréntesis de cierre.

Para resolver este tipo de problemas una pila es un método apropiado. La estrategia de solución es el almacenar en la pila todas las llaves de apertura y cada vez que aparece una de cierre obtenemos un elemento de la pila y debe ser una llave de apertura correspondiente a la de cierre.

Cuando se quiere recuperar un elemento de la pila y está vacía, indica que sobran llaves de cierre. Cuando se termina el proceso la pila debe estar vacía, caso contrario, quiere decir que hay más llaves de apertura que de cierre en la expresión a ser analizada.

Esto nos lleva al siguiente código:

```
import java.io.*;
import java.util.*;

/**
 * Ejemplo de manejo de pilas Comprobación de parentisis
 * bien anidados
 *
 * @author Jorge Teran
 * @param args
 *            parentesis y corchetes EjemploPila ()()(((()[]
 */

public class EjemploPila {
    public static void main(String[] args) {
        String cadena = "()()[][";
        String caracter, delapila;
        // definir una pila de cadena
        Stack<String> pila = new java.util.Stack();
        for (int i = 0; i < cadena.length(); i++) {
            caracter = cadena.substring(i, i + 1);

            if ((caracter.compareTo("(") == 0))
                pila.push(")");
            if (caracter.compareTo("[") == 0)
                pila.push("]");
//            System.out
//                    .println("pone " + caracter);
```

```
            if (pila.isEmpty()) {
                System.out
                        .println("Mal Anidada");
                System.exit(0);
            }
            if ((caracter.compareTo(")") == 0)
                    || (caracter.compareTo("]") == 0)) {
                delapila = pila.pop();
//                System.out.println("sale "
//                        + delapila );
                if (delapila.compareTo(caracter) != 0) {
                    System.out
                            .println("Mal anidada");
                    System.exit(0);
                }
            }
        }

        if (!pila.isEmpty())
            System.out
                    .println("Mal Anidada");
        else
            System.out.println("Bien anidada");
    }
}
```

Los métodos disponibles para pilas son:

Método	Descripción
push(E elemento)	Agrega un elemento a la pila.
pop()	Extrae el último elemento introducido.
peek()	Lee el último elemento introducido.
isEmpty()	Devuelve true si la pila está vacía.

7.4. Listas enlazadas

Las listas enlazadas tienen la ventaja de que permiten ingresar elementos en cualquier lugar de la lista sin necesidad de tener que hacer un espacio como en los vectores. En una lista enlazada existen varias posiciones en las que se puede introducir un elemento, supongamos la lista que tiene los elementos *abc* existen cuatro lugares donde se pueden insertar los nuevos valores, estos son: $\|abc$, $a\|bc$, $ab\|c$ y $abc\|$.

El lenguaje Java provee una clase denominada iterador que permite recorrer la clase. Cuando insertamos elementos consecutivos se introducen en el orden que se envían. Cuando el iterador apunta al principio se introducen al principio, si está apuntando al final de la lista se introducen después del último elemento. El interador, es el que indica donde se puede introducir el elemento.

Los métodos de Java para listas enlazadas son los siguientes:

Método	Descripción
add(E elemento)	Agrega un elemento a la lista.
add(int i, E elemento)	Agrega un elemento en la posición i.
addFirst(E elemento)	Agrega un elemento al principio.
addLast(E elemento)	Agrega un elemento al final.
clear()	Borra la lista.
addAll(int i,coleccion)	Agrega toda una colección en la posición i.
remove(int i)	Quita y devuelve el elemento de la posición i.
removeFirst()	Quita y devuelve el primer elemento.
removeLast()	Quita y devuelve el último elemento.
set(int i, E elemento)	Cambia el elemento de la posición i.
isEmpty()	Devuelve true si la lista esta vacía.
contains(E elemento)	Devuelve true si la lista contiene el elemento.
size()	Devuelve el número de elementos.

Hay que hacer notar que las listas no son una estructura de datos adecuadas para el acceso aleatoreo. Cada vez que se busca un elemento o recorre la lista siempre comienza desde el principio.

La colección para listas enlazadas provee una clase para recorrer la lista llamada iterador que tiene los siguientes métodos:

Método	Descripción
ListIterator(E)	Crea un iterador para visitar elementos de la lista.
set(E elemento)	Cambia el último elemento visitado.
hasPrevious()	Devuelve true si tiene un elemento anterior.
previous()	Devuelve el elemento anterior.
nextIndex()	Devuelve el índice que sería llamado en la próxima iteración.

En el siguiente ejemplo se crean dos listas y se introducen elementos de la clase Alumno. Luego se utiliza el iterador para recorrer la lista. Finalmente se muestra el método `removeAll` que permite remover todos los elementos de una lista que están en la otra.

```
import java.util.*;

/**
 * Ejemplo de lista enlazada
 *
 * @author Jorge Teran
 * @param args
 *            none
 */

public class ListaEnlazada {
    public static void main(String[] args) {
        Alumno alm;
        LinkedList<Alumno> uno = new LinkedList<Alumno>();
        uno.add(new Alumno("Juan", 11));
        uno.add(new Alumno("Maria", 17));

        ListIterator<Alumno> aIter = uno
                .listIterator();

        // desplegar la lista uno
        System.out.println("Listar la lista UNO");
```

```
        while (aIter.hasNext()) {
            alm = aIter.next();
            System.out.println(alm.verNombre());
        }

        List<Alumno> dos = new LinkedList<Alumno>();
        dos.add(new Alumno("Jose", 25));
        dos.add(new Alumno("Laura", 8));
        ListIterator<Alumno> bIter = dos
                .listIterator();

        // agregar los alumnos de lista dos
        // a la lista uno
        while (bIter.hasNext()) {
            uno.add(bIter.next());
        }
        System.out
                .println("Despues de agregar la lista DOS");
        // Reiniciar el iterador y listar la lista uno
        aIter = uno.listIterator();
        while (aIter.hasNext()) {
            alm = aIter.next();
            System.out.println(alm.verNombre());
        }

        // quitar los elementos de la lista dos de uno
        uno.removeAll(dos);

        System.out
             .println("Listar despues de quitar la lista DOS");
        // Reiniciar el iterador y listar la lista uno
        aIter = uno.listIterator();
        while (aIter.hasNext()) {
            System.out.println(aIter.next()
                    .verNombre());
        }
    }
}
```

Para realizar listas doblemente enlazadas es suficiente crear una lista con una clase de tipo lista. De esta forma se pueden crear listas con cualquier nivel de anidamiento.

7.5. Conjuntos

El manejo de conjuntos es similar, al de las listas con la condición que, no se permiten elementos repetidos y el orden no es relevante.

La implementación de un conjunto se hace como una estructura de tipo `Set` y esta estructura viene implementada en `HashSet`. Para definir un conjuto `conjA` hay que escribir lo siguiente:

```
Set<String> conjA = new HashSet<String>();
```

Los métodos disponibles para la clase set son los siguientes:

Método	Descripción
add(E elemento)	Agrega un elemento al conjunto.
clear()	Borra el conjunto.
addAll(coleccion)	Agrega toda una colección.
removeAll(coleccion)	Elimina todos los elementos de la colección que existan en el conjunto.
remove(Object objeto)	Elimina el objeto del conjunto. Devuelve true si ha sido eliminado.
isEmpty()	Devuelve true si el conjunto esta vacío.
contains(E elemento)	Devuelve true si el conjunto contiene el elemento.
size()	Devuelve el número de elementos del conjunto.

Los métodos para recorrer los conjuntos se realizan con la clase `Iterator` que tiene los siguientes métodos:

Método	Descripción
Iterator(E)	Crea un iterador para visitar elementos del conjunto.
hasNext()	Si existe un próximo elemento devuelve true.
next()	Avanza al próximo elemento.

Para utilizar la clase `Iterator` se procede como sigue:

- Primero creamos el iterador sobre el conjunto

  ```
  Iterator<String> iter = conjA.iterator();
  ```

- Para recorrer el conjunto

  ```
  while (iter.hasNext())
  ```

- Finalmente para acceder a elementos individuales podemos utilizar `iter.next()`. Por ejemplo para desplegar el elemento:

  ```
  System.out.println(iter.next());
  ```

En el siguiente ejemplo se muestra como introducir elementos, recorrer el conjunto y como hallar la intersección de dos conjuntos.

```
import java.util.*;
/**
 * Ejemplo manejo de conjuntos
 * @author Jorge Teran
 * @param args
 *            none
 */
public class Conjuntos {
  public static void main(String[] args) {
    Set<String> conjA = new HashSet<String>();

    conjA.add("aaa");
    conjA.add("bbb");
    conjA.add("aaa");
    conjA.add("ccc");
    conjA.add("ddd");

    Set<String> conjB = new HashSet<String>();

    conjB.add("aaa");
```

```
    conjB.add("bbb");
    conjB.add("bbb");
    conjB.add("xxx");
    conjB.add("yyy");
    //hallar conjB interseccion conjA
    Set<String> conjC = new HashSet<String>();
    conjC.addAll(conjA);
    // para intersectar A y B
    // hacemos C=A-B y luego A-C
    conjC.removeAll(conjB);
    conjA.removeAll(conjC);
    //listar
    Iterator<String> iter = conjA.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());
  }
}
```

7.6. Clases map

Las clases donde se almacenan los valores como clave y valor se denomina clase *Map* por ejemplo, si se quiere almacenar *color=rojo* donde *color* es la clave y *rojo* es el valor se puede almacenar y recuperar por la clave.

Los métodos de la clase `Map` están descriptos en el cuadro 7.1.

El recorrido de la clase Map se puede realizar por clave, valor o por ambos. Para definir un recorrido de una clase Map por clave se procede como sigue:

```
Iterator<String> iter = preferencias.keySet().iterator();
```

Para definir un recorrido por valor:

```
Iterator<String> iter = preferencias.values().iterator();
```

Para recorrer por ambos hay que tomar en cuenta que `Iterator` no permite definir dos cadenas como es la definición de Map. Por esto es necesario cambiar el recorrido como sigue:

```
Map.Entry<String, String> datos : preferencias.entrySet()
```

Método	Descripción
put(Clave, Valor)	Agrega un elemento Map.
get(Clave)	Devuelve el elemento Map con la clave especificada.
clear()	Borra el conjunto Map.
remove(Object objeto)	Elimina el objeto del conjunto con la clave especificada. Devuelve true si es eliminado.
isEmpty()	Devuelve true si el conjunto Map esta vacío.
containsValue(Object elemento)	Devuelve true si el conjunto contiene el valor.
containsKey(Object elemento)	Devuelve true si el conjunto contiene la clave.
size()	Devuelve el número de elementos del conjunto Map.

Cuadro 7.1: Métodos de la clase map

Luego podemos acceder a la clave con `datos.getKey()` y a los valores a través de `datos.getValue()`.

En el siguiente ejemplo, se introducen pares de elementos y luego se recorre el conjunto Map listando clave y valor.

```
import java.util.*;
/**
 * Ejemplo de la estructura map
 * @author Jorge Teran
 * @param args
 *            none
 */
public class PruebaMap {
  public static void main(String[] args) {
    Map<String, String> preferencias =
                new HashMap<String, String>();
    preferencias.put("color", "rojo");
    preferencias.put("ancho", "640");
    preferencias.put("alto", "480");
```

```
    //listar todo
    System.out.println(preferencias);

    // Quitar color

    preferencias.remove("color");

    // cambiar una entrada

    preferencias.put("ancho", " 1024");

    // recuperar un valor

    System.out.println("Alto= "
              + preferencias.get("alto"));

    // iterar por todos los elementos

    for (Map.Entry<String, String> datos :
                           preferencias.entrySet()) {
      String clave = datos.getKey();
      String valor = datos.getValue();
      System.out.println(
               "clave=" + clave + ", valor=" + valor);
    }
  }
}
```

7.7. Clases para árboles

En el Java tenemos la clase de árboles que permite mantener ordenados los elementos de un conjunto a medida que, se introducen los elementos. Para poder implementar ésto es necesario tener una clase de comparación que permita decidir en que rama del árbol corresponde insertar un elemento, para esto agregamos a la clase Alumno el siguiente método.

```
public int compareTo(Alumno other) {
   return código - other.código;
```

```
   }
```

que superpone el método de comparación que solo se utiliza con los tipos de datos básicos; vea que, lo que se está comparando es el código y la respuesta puede ser 0,1 o -1. En la definición de la clase es necesario agregar `implements Comparable<Alumno>` quedando como sigue:

```
public class Alumno implements Comparable<Alumno> {
```

Luego el siguiente código ingresa Alumnos a un árbol organizado por código

```
import java.util.*;
/**
 * Ejemplo de la estructura de Arboles
 * @author Jorge Teran
 * @param args
 *            none
 */
public class Arbol {
  public static void main(String[] args) {
    TreeSet<Alumno> alm = new TreeSet<Alumno>();
    alm.add(new Alumno("Juan", 11));
    alm.add(new Alumno("Maria", 17));
    alm.add(new Alumno("Jose", 25));
    alm.add(new Alumno("Laura", 8));
    System.out.println(alm);
}
```

No siempre es el único orden que deseamos tener, por ejemplo si se desea ordenar por nombre, no es posible especificar dos clases de comparación. Por este motivo podemos crear otro árbol especificando su clase de comparación.

```
TreeSet<Alumno> ordenarPorNombre = new TreeSet<Alumno>(
     new Comparator<Alumno>() {
        public int compare(Alumno a, Alumno b) {
        String nombreA = a.verNombre();
        String nombreB = b.verNombre();
        return nombreA.compareTo(nombreB);
       }
     });
```

```
      ordenarPorNombre.addAll(alm);
      System.out.println(ordenarPorNombre);
  }
}
```

La instrucción `ordenarPorNombre.addAll(alm)` permite insertar todo el árbol *alm* en el árbol *ordenarPorNombre*. Los métodos de la clase *TreeSet* son los siguientes:

Método	Descripción
add(E elemento)	Agrega un elemento al árbol.
clear()	Borra el árbol.
addAll(coleccion)	Agrega toda una colección.
remove(Object objeto)	Elimina el objeto del árbol. Devuelve true si ha sido eliminado
isEmpty()	Devuelve true si el árbol está vacío.
contains(E elemento)	Devuelve true si el conjunto contiene el elemento.
size()	Devuelve el número de elementos del árbol.

Los métodos para recorrer el árbol elemento por elemento se realiza con la clase `Iterator` especificando que el iterador es alumno y se procede como sigue:

```
Iterator<Alumno> iter = alm.iterator();
   while (iter.hasNext())
     System.out.println(iter.next());
```

7.8. Clases para colas de prioridad

Las colas de prioridad tienen la característica de estar basadas en la estructura de datos denominada montículo. La característica principal es que cada elemento introducido está asociado a una prioridad. A medidad que se van introduciendo valores se van organizando para que al recuperar uno, siempre se recupere primero, el que tiene menos prioridad.

La aplicación típica de ésta cola es por ejemplo el organizar una lista de impresión por prioridad a medida que se introducen y procesan trabajos, siempre se procesa primero el de menor prioridad.

Para ejemplificar el uso en Java presentamos el siguiente ejemplo que muestra como se introducen y obtienen uno a uno lo elementos comenzando del valor mínimo.

```
import java.util.*;

/**
 * Ejemplo de Cola de prioridad
 *
 * @author Jorge Teran
 * @param args
 *            none
 */

public class ColaPrioridad {
    public static void main(String[] args) {
        PriorityQueue<Alumno> cp = new PriorityQueue<Alumno>();
        cp.add(new Alumno("Juan", 11));
        cp.add(new Alumno("Maria", 17));
        cp.add(new Alumno("Jose", 25));
        cp.add(new Alumno("Laura", 8));

        System.out
                .println("Sacando elementos...");
        Alumno datos;
        while (!cp.isEmpty()) {
            datos = cp.remove();
            System.out.println(datos.verCodigo()
                    + " = " + datos.verNombre());
        }
    }
}
```

Analizando el código vemos que los datos ingresados serán extraídos según la prioridad. En este ejemplo, hemos tomado como prioridad el código, dado que a la clase Alumno se ha adicionado el método *comparTo* del ejemplo anterior que utiliza el campo código en la comparación. Por esto, la salida del programa está en el orden 8, 11, 18, 25.

Método	Descripción
add(E elemento)	Agrega un elemento a la cola de prioridad.
clear()	Borra la cola de prioridad.
addAll(coleccion)	Agrega toda una colección.
remove()	Elimina el elemento más pequeño.
isEmpty()	Devuelve true si está vacía.
contains(E elemento)	Devuelve true si el conjunto contiene el elemento.
size()	Devuelve el número de elementos.

Cuadro 7.2: Métodos de las colas de prioridad

Los métodos disponibles para las colas de prioridad están descriptos en el cuadro 7.2.

7.9. Ejercicios para laboratorio

Los siguientes ejercicios de laboratorio tienen el objetivo de determinar en la práctica las estructuras más eficientes para resolver diferentes problemas. Para esto se deben tomar los tiempos de ejecución de cada estructura de datos, construir la siguiente tabla y analizar los resultados.

Estructura	Tiempo de ejecución
Vector	
ArrayList	
Pila	
Lista enlazada	
Arbol	
Cola de prioridad	
Conjuntos	
Map	

1. Hallar el tiempo de insertar 10.000 datos en cada una de las estructuras. Explicar la ventaja de cada una de ellas.

2. Insertar 10.000 números aleatorios a cada una de las estructuras presentadas en el capítulo y encontrar cual es más eficiente para hallar los 100 más pequeños.

3. En una combinación de operaciones: 10 % recorrer ordenado, 30 % buscar, 40 % insertar y 20 % remover un valor arbirario. Realice un ejercicio con 10.000 valores y analice tres estructuras de datos. Escoja las más adecuadas.

4. Repetir el ejercicio tres para 10^3, 10^4, 10^5 y 10^6 valores. Indicar en qué casos es más conveniente cada estructura.

7.10. Ejercicios para el Juez de Valladolid

107, 115, 122, 126, 327, 496, 514, 536, 673, 679, 699, 712, 727, 793, 912, 10152, 10158, 10185, 10226, 10273, 10493, 10505, 10614, 10666, 10679, 10909, 10935, 10938, 11032, 11103, 11111, 11234, 11235, 11239, 11291, 11297, 11354, 11402, 11423, 11465, 11536, 11537, 11601, 11615, 11678, 11714.

0001

Capítulo 8

Backtracking

8.1. Introducción

Backtracking significa ir atrás. Esto implica llevar un registro de los eventos pasados, con el propósito, de realizar un proceso más inteligente. Aunque este método aparenta una búsqueda exhaustiva, no examina todas las combinaciones posibles.

Este tema es introducido mostrando el recorrido de un grafo que muestra el concepto de volver a un estado anterior.

Posteriormente se trabaja construyendo subconjuntos y permutaciones. Para este tipo de problemas se desarrolla una implementación básica que consiste de tres partes: una que determina si hemos hallado una solución, la segunda que procesa la solución y la tercera que realiza el proceso.

8.2. Recorrido de grafos

Un ejemplo típico de algoritmo de retroceso es el hallar un camino desde un origen hasta un destino en grafo. Consideremos el grafo de la figura 8.1:

Si partimos del nodo A y queremos llegar al nodo G tenemos dos estrategías para hallar un camino. Una de ellas es hacer todas las combinaciones posibles partiendo del origen. La segunda es empezar a recorrer el grafo retrocediendo un lugar cada vez que se llegue a un nodo del que no podamos avanzar más.

El proceso comienza en nodo A, y se va al primer nodo que encontramos el nodo B. Ver figura 8.2.

Figura 8.1: Ejemplo de un grafo

Figura 8.2: Muestra el recorrido del nodo A a B

Del nodo B se procede hasta el nodo E que es el primer nodo que encontramos.

No hay forma en éste momento de elegir el nodo F porque no sabemos si nos lleva al destino. De esta forma llegamos al nodo C y posteriormente a D. Ver figura 8.3.

Cuando llegamos al nodo D ya no se puede seguir (figura 8.4), por lo que, es necesario regresar al nodo anterior C y vemos que no hay ningún otro nodo para visitar, retrocedemos una vez más llegando a E y continuamos.

Cuando llegamos al punto final imprimimos el recorrido. Cuando el algoritmo llega al inicio y no hay nodos que no hayamos visitado termina sin encontrar un camino.

Para programar hemos construido una matriz para representar el grafo. En esta matriz las filas y columnas representan los nodos y la posición *(i,j)* indica que existe un camino cuando tiene un *1* y un 0 cuando no existe un camino. Para el grafo de ejemplo la matriz que constuimos es:

Figura 8.3: Muestra el recorrido hasta el nodo C

Figura 8.4: Muestra el proceso de retorceso

	A	B	C	D	E	F	G
A	0	1	0	0	1	0	0
B	0	0	0	0	1	1	0
C	0	0	0	1	0	0	0
D	0	0	0	0	0	0	0
E	0	0	1	0	0	1	0
F	0	0	0	1	0	0	1
G	0	0	0	0	0	0	0

Para el proceso se comienza con la posición 0. Cada vez que pasamos por un nodo lo marcamos como visitado. Esto es para evitar ciclos y volver a realizar un recorrido anterior. Cuando se llega a un nodo que no va a ningún lado regresamos (bactrack) al nodo anterior para buscar una nueva ruta. El nodo anterior se ha guardado en una estructura de pila.

El programa recorre el grafo mostrado y produce la ruta en orden inverso listando: 6 5 4 1 0.

```
/**Programa para mostrar el
  * recorrido de un grafo
  * @author Jorge Teran
 */
```

```
import java.util.Stack;
import java.util.Vector;
public class Grafo2 {
  static void Camino(int[][] g) {
    int i = 0;
    Stack<Integer> pila = new Stack();
    // visitados
    int[] v = new int[g[0].length];
    i = 0;
    while (v[v.length - 1] == 0) {
      pila.push(i);
      v[i] = 1;
      i = Buscar(g, v, i);
      if ((i == -1) && (v[v.length - 1] == 0)) {
        if (pila.empty()) {
          System.out.println("No Hay Ruta ");
          System.exit(0);
        } else {
          //bactracking al anterior
          i = (int) pila.pop();
          i = (int) pila.pop();
        }
      }
    }
    System.out.println("Ruta en Orden Inverso ");
    while (!pila.empty()) {
      i = (int) pila.pop();
      System.out.print(" " + i);
    }

  }

  static int Buscar(int[][] g, int[] v, int fila) {
    int hallo = -1, i;
    for (i = 0; i < g[0].length; i++) {
      // System.out.println("busca ="+g[fila][i]);
      if ((g[fila][i] == 1) && (v[i] == 0)) {
        hallo = i;
```

```
        break;
      }
    }
    // System.out.println("Hallo ="+hallo);
    return hallo;
  }

  public static void main(String[] args) {
    int[][] g = new int[7][7];
    // Nodos A=0,B=1,C=2,D=3,E=4,F=5,G=6
    g[0][1] = 1;
    g[0][4] = 1;
    g[1][4] = 1;
    g[1][5] = 1;
    g[2][3] = 1;
    g[3][0] = 0;
    g[4][2] = 1;
    g[4][5] = 1;
    g[5][3] = 1;
    g[5][6] = 1;
    g[6][0] = 0;
    Camino(g);
  }
}
```

8.3. Constuir todos los subconjuntos

Consideremos el problema de construir todos los subconjuntos de un conjunto de tamaño n. El número total de subconjuntos es $2^n - 1$ sin contar el conjunto vacío. Por ejemplo si $n = 3$ existen 7 subconjuntos.

Consideremos el conjunto {1,2,3} los 7 posibles subconjuntos son: {1 2 3 }, {1 2 }, {1 3 }, {1 }, {2 3 }, {2 } y {3 }.

Para construir un programa que construya éstos subconjuntos inicialmente se construye un vector de unos y ceros para representar las posibles soluciones. Al inicio de cada recursión preguntamos si hemos hallado una solución para procesar. Si no se halló una solución guardamos el vector de soluciones y probamos una nueva solución. En este problema hemos considerado una

solución cuando llegamos al último elemento del conjunto.

Luego regresamos a un estado anterior y comenzamos la búsqueda de una nueva solución. La instrucción `subset[cur] = true;` se usa para indicar que este elemento puede ser parte de la solución. Luego se procesa recursivamente una nueva solución.

Cuando hallamos una solución se retorna al estado anterior y marcamos el último elemento como procesado. EL código que halla esta solución es el siguiente:

```
/**Programa para hallar todos los
  * subconjuntos
  * @author Jorge Teran
 */
public class Subconjuntos {

  private static void procesarSolucion(
                          boolean[] conjuntos) {

    // imprimir la solucion,
    // conjuntos indica cuales imprimir
    System.out.print("{");
    for (int i = 0; i < conjuntos.length; i++)
      if (conjuntos[i])
        System.out.print((i + 1) + " ");

    System.out.println("}");
  }

  public static boolean esSolucion(
            boolean[] conjuntos, int posicionActual) {
    // cuando se llego al final del vector
    // no se puede seguir iterando
    return (conjuntos.length == posicionActual);
  }

  public static void imprimeSubconjuntos(
           boolean[] conjuntos,int posicionActual) {
```

```
    if (esSolucion(conjuntos, posicionActual)) {
      procesarSolucion(conjuntos);
    } else {
      conjuntos[posicionActual] = true;
      // procesar una nueva solucion
      imprimeSubconjuntos(conjuntos, posicionActual + 1);
      // retornar (bactrack) al caso anterior
      conjuntos[posicionActual] = false;
      imprimeSubconjuntos(conjuntos, posicionActual + 1);
    }
  }

  public static void main(String[] args) {
    boolean[] conjuntos = new boolean[3];
    // Inicialmente conjuntos esta todo en false
    imprimeSubconjuntos(conjuntos, 0);
  }

}
```

Al analizar el proceso se observa que se procede a marcar las posibles soluciones cambiando el arreglo que hemos denominado *conjuntos* cambiando el estado de 000 a 100, 110 y 111. Cuando se llega a marcar tres elementos se imprime la primera solución {1, 2, 3 }.

En éste momento la pila de la recurción contiene los valore 000, 100, 110 y 111. En el proceso recursivo la pila es la que guarda los resultados previos.

Ahora se produce un retroceso, el arreglo que almacena los estados contiene 111 que se recupera de la pila. Luego se cambia el estado del último bit a 0 para indicar que ya fue procesado y volvemos a llamar recursivamente al programa almacenando en la pila el 110, que produce la solución {1, 2 }. Al terminar sale este valor de la pila y regresa al valor almacenado 100. Se procede a marcar el 101 el bit 2 y se continúa el proceso. El proceso del programa se muestra en el cuadro 8.1. Analizando la tabla, vemos que en la columna de la izquierda representa un contador binario. Usado este concepto, creamos un contador binario y luego imprimimos todos los elementos del conjunto que corresponden a un bit 0.

En el ejemplo, el número binario 011, representa el subconjunto $\{2, 3\}$. El elemento $\{1\}$ no se imprime porque el bit correspondiente está en 0. Con

Bits de Estado	Posición Actual	Retroceso	Solución
100	0		
110	1		
111	2		{1, 2, 3 }
111	2	Fin Recursión 1	
110	2		{1, 2 }
110		Fin Recursión 2	
100	1	Fin Recursión 1	
101	2		{1, 3 }
100	2	Fin Recursión 1	
100	2		{1 }
100	2	Fin Recursión 2	
100	1	Fin Recursión 2	
000	0	Fin Recursión 1	
010	1		
011	2		{2,3}
010	2	Fin Recursión 1	
010	2		{2}
010	2	Fin Recursión 2	
000	1	Fin Recursión 1	
001	2		{3}
		Fin Recursión 2	

b

Cuadro 8.1: Proceso realizado para construir todos los subconjuntos

estas consideraciones podemos escribir un programa mucho más simple.

```
public class SubConj {
    /* Programa para hallar todos
     * los subconjuntos
     * @author Jorge Teran
     */
    public static void main(String[] args) {
        char [] conjunto = {'1','2','3'};
        int nbits = conjunto.length;
        int max = 1 << conjunto.length;
        for(int i = 0; i < max ;i++) {
            for (int j = 0; j<nbits; j++){
                if ((i & (1<<j)) >0)
                    System.out.print(conjunto[j]);
            }
            System.out.println();
        }
    }
}
```

Compare la salida de este programa

$$\{1\},\ \{2\},\{1,2\},\ \{3\},\{1,3\},\{2,3\},\{1,2,3\}$$

Solamente el orden es diferente. ¿Puede modificar el programa para construir el mismo orden? Con éste análisis hemos mejorado el algoritmo recursivo para utilizar mucho menos memoria y tiempo de proceso.

8.4. Construir todas las permutaciones

Para construir todas las permutaciones de un conjunto también se utiliza la técnica de backtrackig. En este ejemplo se ha construído un programa similar al anterior con una clase que procesa la solución y otra que determina si ha llegado la solución.

El programa intercambia valores empezando del primer valor en forma recursiva al igual que el programa de hallar todos los subconjuntos. Comienza intercambiando el primer elemento con el último, en un ciclo. Recursivamente retornando al estado anterior cuando se halla una permutación.

```
/**Programa para hallar todas
 * las permutaciones
 * @author Jorge Teran
*/

public class Permutaciones {
//  imprimir las n! permutaciones en desorden
  private static void procesarSolucion(String[] a) {
    for (int i = 0; i < 3; i++) {
      System.out.print(a[i]);
    }
    System.out.println(" ");
  }
  public static boolean esSolucion(int posicionActual) {
    // cuando se llego al final del vector
    // no se puede seguir iterando
    return (posicionActual == 1);
  }

  private static void permutar(
              String[] a, int n) {
    if (esSolucion(n)) {
      procesarSolucion(a);
      return;
    }
    for (int i = 0; i < n; i++) {
      swap(a, i, n-1);
      permutar(a, n-1);
      swap(a, i, n-1);
    }
  }
  private static void swap(String[] a, int i, int j) {
    String c;
    c = a[i]; a[i] = a[j]; a[j] = c;
  }
  public static void main(String[] args) {
    String[] a = new String[3];
    a[0]="a";
```

```
    a[1]="b";
    a[2]="c";
    permutar(a,3);
  }
}
```

El resultado que se obtiene del programa es: bca, cba, cab, acb, bac, abc.

Como se ve en el ejemplo anterior los resultados no están en orden. En orden significa que los resultados deben estar en la siguiente secuencia: abc, acb, bac, bca, cab, cba. Esto significa realizar todos los intercambios manteniendo el primer elemento, luego llevar el segundo elemento al principio y así sucesivamente.

Para realizar ésto hemos utilizado una estructura de vector que, permite insertar y extraer objetos. Se crearon dos vectores, uno para almacenar las combinaciones que se van formando y otro que mantiene el vector original. El siguiente código permite realizar estas permutaciones.

```
/**Programa para hallar todas
 * las permutaciones en orden
 * @author Jorge Teran
*/
import java.util.Vector;
public class PermutacionesEnOrden {

  private static void procesarSolucion(Vector a) {
    System.out.println(a);
  }

  public static boolean esSolucion(int posicionActual) {
    // cuando se llego al final del vector
    // no se puede seguir iterando
    return (posicionActual == 0);
  }

  private static void permutar(
              Vector auxiliar, Vector original) {
    int N = original.size();
    if (esSolucion(N)) {
```

```
      procesarSolucion(auxiliar);
    } else {
      for (int i = 0; i < N; i++) {
        String proximo = (String) original.remove(i);
        auxiliar.add(proximo);
        permutar(auxiliar, original);
        original.add(i, proximo);
        auxiliar.remove(proximo);
      }
    }
  public static void main(String[] args) {
    Vector v = new Vector();
    String[] a = new String[3];
    a[0] = "a";
    a[1] = "b";
    a[2] = "c";
    for (int i = 0; i < a.length; i++) {
      String s = a[i];
      v.add(s);
    }
    permutar(new Vector(), v);
  }
}
```

8.5. Ejemplo de Aplicación

Presentamos el ejercicio 729 del Juez de Valladolid:

La distancia de Hamming entre dos números binarios es el número de bits que difieren. Esto se puede hallar utilizando la instrucción *XOR* en los bits correspondientes. Por ejemplo en las dos cadenas de bits siguientes:

A		0	1	0	0	1	0	1	0	0	0
B		1	1	0	1	0	1	0	1	0	0
A XOR B	=	1	0	0	1	1	1	1	1	0	0

La distancia de Hamming (H) entre estas dos cadenas de 10-bits es de 6, el número de bits en uno después de la instrucción *XOR*.

Entrada La entrada consiste en varios casos de prueba, la primera línea de la entrada contiene el número de casos de prueba seguida de una línea en blanco. Las siguientes líneas consisten de N la longitud de la cadena de bits, y la distancia de Hamming H separados por un espacio ($1 \leq N, H \leq 16$).

Salida Para cada dato de entrada liste todas las posibles cadenas de longitud N que tienen una distancia de Hamming H. Esto es todas las cadenas de longitud N con exactamente H números 1, impreso en orden lexicográfico.

El número de estas cadenas es igual a la combinatoria $C(N, H)$. Esto es el número de combinaciones posibles de $N - H$ ceros y H unos. Esto es igual a:

$$\frac{N!}{(N-H)!H!}$$

Imprima una línea en blanco entre conjuntos de datos.

```
Ejemplo de entrada
1

4 2

Ejemplo de Salida
0011
0101
0110
1001
1010
1100
```

Estrategias de solución Para resolver el problema podemos tomar varias estrategias de solución, vemos el ejemplo. Con cuatro bits puedo tener 2^4 posibles cadenas que son: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 0011, 1100, 1101, 1110, 1111. De esta lista podemos escoger las cadenas que tienen dos unos.

Claro está que no podemos escribir todas las cadenas porque todas las posibles para 16 tomadas de 2 es un número muy grande. Podemos divisar dos alternativas, la primera puede ser escribir una cadena con la cantidad de 1, y 0 requeridos, en el ejemplo seria 0011 y luego hacer las combinaciones de los dígitos para obtener 0101, 0110, 1001, 1010 y 1100. La segunda solución se puede construir convirtiendo las cadenas a números decimales. Podemos ver que todas las combinaciones dan los números $0, 1, 2, 3, 4, 5..., 16$ que se pueden construir con un contador. Luego contamos los bits que son 1 y si corresponden a la distancia Hamming imprimimos el número en binario.

El código siguiente corresponde a la solución.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        int ncasos, nbits, dhamming;
        int i = 0;
        int c = 0, n = 0, nmax = 0;
        char[] salida;
        Scanner lee = new Scanner(System.in);
        ncasos = lee.nextInt();
        while (0 < ncasos--) {
            nbits = lee.nextInt();
            dhamming = lee.nextInt();
            nmax = 1 << nbits;
            for (int j = 1; j <= nmax; j++) {
                n = j;
                c = 0;
                //contar cuantos bits son 1
                for (int l = 0; l < nbits; l++) {
                    if ((n & 1) == 1) {
                        c++;
                    }
```

```
                n = n >> 1;
            }

            if (c == dhamming) {
                salida = new char[nbits];
                i = 0;
                //transformar a binario
                for (int l = nmax / 2; l > 0; l = l >> 1) {
                    if ((l & j) == 0)
                        salida[i] = '0';
                    else
                        salida[i] = '1';
                    i++;
                }
                System.out.println(salida);
            }
        }
        if (ncasos > 0)
            System.out.println("");
    }
  }
}
```

Una solución del problema también puede realizarse utilizando las funciones del lenguaje que nos permiten convertir un número a binario y contar el número de bits que están en 1. El método *bitCount* de la clase *Integer* nos da el número de bits que son 1. Comparando con la distancia de Hamming pedida ya tenemos la solución.

El método *toBinaryString* de la clase *Integer* permite convertir un número entero a su representación binaria. Como elimina los ceros de la izquierda solo es necesario completar los mismos para obtener la respuesta. El código siguiente muestra esta versión de la solución.

```
import java.util.Scanner;

public class P729j {
    public static void main(String[] args) {
        int ncasos, nbits, dhamming;
```

```
        int nmax = 0;
        String ceros = "00000000000000000000", salida = "";
        Scanner lee = new Scanner(System.in);
        ncasos = lee.nextInt();
        while (0 < ncasos--) {
            nbits = lee.nextInt();
            dhamming = lee.nextInt();
            nmax = 1 << nbits;
            for (int j = 1; j < nmax; j++) {
                if (Integer.bitCount(j) == dhamming) {
                    salida = Integer
                            .toBinaryString(j);
                    if (salida.length() < nbits)
                        salida = ceros
                      .substring(0, nbits - salida.length())
                             + Integer.toBinaryString(j);
                    System.out.println(salida);
                }
            }
            if (ncasos > 0)
                System.out.println("");
        }
    }
}
```

8.6. Ejercicios

Resolver los ejercicios del Juez de Valladolid:

110,112, 140, 148, 167, 347, 441, 574, 604, 639, 729, 750, 861, 989, 10001, 10012, 10017, 10094, 10123, 10150, 10160, 10186, 10344, 10508, 10624, 10637, 11127, 11201, 11218, 11325, 11390, 11587, 11602, 11659.

Capítulo 9

Geometría computacional

9.1. Introducción

La geometría computacional trata de una serie conceptos que se definen a continuación:

Triangulación de polígonos Triangulación es la división de un polígono plano en un conjunto de triángulos, usualmente con la restricción de que cada lado del triángulo sea compartido por dos triángulos adyacentes.

Problemas de partición La triangulación es un caso especial de la partición de un polígono. Lo que trata normalmente es de particionar el polígono en objetos convexos más pequeños, porque es más fácil trabajar con objetos pequeños.

Problemas de intersección En el plano los problemas de intersección están relacionados con encontrar los puntos de intersección de todos los segmentos que se forman con los puntos especificados. En dimensiones superiores se analizan también la intersección de poliedros.

Cerco convexo o convex hull Un cerco convexo crea un subconjunto de un plano denominado plano convexo, sí y solo sí cualquier segmento de recta de los puntos especificado está en este subconjunto.

Búsqueda geométrica La búsqueda geométrica se refiere a tres problemas principales: ubicación de un punto en un polígono, visibilidad para conocer las áreas visibles al iluminar con un rayo de luz y las búsquedas en un subconjunto específico.

Areas de polígonos Comprende los métodos para calcular el área de un polígono, dada una lista de puntos.

Una descripción de las temáticas que trata el campo de la geometría computacional se puede encontrar en [Cha94].

Iniciaremos el desarrollo de éste capítulo con concepto de geometría y luego se describirá la geometría computacional. En ambos casos se mostrarán algunos algoritmos y la implementación de las librerías de Java.

9.2. Geometría

La geometría es el estudio matemático de elementos en el espacio, por ejemplo puntos, líneas, planos y volúmenes. Aún cuando esta geometría ha sido estudiada ampliamente se quiere mostrar como se implementa computacionalmente y cómo un conjunto básico de métodos nos ayuda a resolver una variedad de problemas si tener que resolver cada uno, reduciendo significativamente el tiempo de desarrollo. Una buena bibliografía para temas de geometría se puede leer en [Leh88].

Bien para definir un punto crearemos la clase *Punto* con dos parámetros el valor del eje x y el valor del eje y. El código para definir este objeto es:

```
/* Clase para definir un punto
 * @author Jorge Teran
 */

  public Punto(double x, double y) {
    super();
    this.x = x;
    this.y = y;
  }
```

Los valores se han definido como *double* para permitir la implementación de los métodos que efectúan cálculos cuyos resultados no son enteros.

En el desarrollo de ésta clase se han definido una serie de métodos generales que nos permiten resolver una variedad de problemas. Tomando los conceptos de la geometría, se presenta la distancia a otro punto y el área de un triángulo que está formada por tres puntos.

El resultado da la siguiente clase:

```
/* Clase para definir un punto
 * @author Jorge Teran
 */

public class Punto {
  double x, y;

  public Punto(double x, double y) {
    super();
    this.x = x;
    this.y = y;
  }

  double X() {
    return (x);
  }

  double Y() {
    return (y);
  }

  public String toString() {
    return ("(" + x + "," + y + ")");
  }

  double Distancia(Punto p) {
    double d = 0;
    d = (x - p.X()) * (x - p.X())
        + (y - p.Y()) * (y - p.Y());
    d = Math.sqrt(d);
    return d;
  }

  double areaTriangulo(Punto b, Punto c) {
    double d = 0;
    d = x * b.X() - y * b.X() + y * c.X()
        - x * c.Y() + b.X() * c.Y()
        -c.X() * b.Y();
```

```
    d = Math.abs(d / 2);
    return d;
  }
}
```

Para definir una recta en el plano existen varias alternativas, se puede utilizar la expresión $y = ax + b$ sin embargo tiene el problema que cuando la recta es paralela al eje y dado que la pendiente es infinita haciendo que haya que tomar algunos recaudos.

Existe una representación más apropiada que es $ax + by + c = 0$ que es la que se construyó en este ejemplo. También puede definirse la recta con dos puntos. En la clase que se ha construido se ha tomado las dos anteriores y la que se define por un punto más una pendiente. Cuando se dan los dos puntos construímos los valores a, b, c desde ésta especificación.

A fin de poder comparar dos números no enteros hemos construido una variable que la hemos llamado *ERROR*. Esto se hace porque las operaciones de punto flotante no dan un valor exacto.

Los métodos implementados son: punto de intersección, prueba de líneas paralelas, pendiente, línea perpendicular que pasa por un punto. La clase completa queda como sigue:

```
public class Linea {
/* Clase para definir una linea
 * @author Jorge Teran
 */

  double a, b, c;

  public Linea(double a, double b, double c) {
    this.a = a;
    this.b = b;
    this.c = c;
  }

  public Linea(Punto p1, Punto p2) {
    this.a = p1.Y() - p2.Y();
    this.b = -p1.X() + p2.X();
    this.c = p1.X() * p2.Y() - p1.Y() * p2.X();
  }
```

```
public Linea(Punto p, double m) {
  double a, b;
  a = m;
  b = 1;
  c = -(a * p.X() + b * p.Y());
}

public static final double ERROR = 0.0001;

double A() {

  return a;
}

double B() {

  return b;
}

double C() {

  return c;
}

Punto Interseccion(Linea l) {
  double x, y;
  if (Paralelas(l)) {
    System.out.println("Error: Lineas parelas");
    // System.exit(0);
  }
  x = (l.B() * c - b * l.C()) / (l.A() * b - a * l.B());

  if (Math.abs(b) > ERROR)
    y = -(a * x + c) / b;
  else
    y = -(l.A() * x + l.C()) / l.B();
```

```
    return (new Punto(x, y));
  }

  Boolean Paralelas(Linea l) {
    return ((Math.abs(a - l.A()) <= ERROR)
             && ((Math.abs(b - l.B()) <= ERROR)));
  }

  double Pendiente() {
    if (b == 0) // pendiente infinita
      return (0);
    else
      return (a / b);
  }

  Linea lineaPerpendicularPunto(Punto l) {
    return (new Linea(-b, a, (b * l.X() - a * l.Y())));
  }

  public String toString() {
    return (a + "x+" + b + "y+" + c + "=0");
  }
}
```

Se pueden construir una variedad de clases para manejar diferentes figuras geométricas que incluyan funciones básicas que, combinadas permiten resolver algunos problemas utilizando sus funciones.

Como ejemplo, consideremos una lista de puntos de la cuál queremos hallar cuáles son los dos puntos más cercanos.

Para resolver ésto definimos un vector de puntos y con dos ciclos compara-

mos exaustivamente todas las posibilidas hallando la distancia mínima. Este código con tiempo de ejecución proporcional a $O(n^2)$ es el siguiente:

```
public class minimoPuntos {
/*
 * @author Jorge Teran
 */

  public static void main(String[] args) {
    Punto[] p = new Punto[5];
    double minimo = 999999, d = 0;
    int punto1 = 0, punto2 = 0;
    p[0] = new Punto(8, 16);
    p[1] = new Punto(8, 11);
    p[2] = new Punto(12, 16);
    p[3] = new Punto(24, 10);
    p[4] = new Punto(13, 8);
    for (int i = 0; i < 5; i++) {
      for (int j = i + 1; j < 5; j++) {
        d = p[i].Distancia(p[j]);
        if (d < minimo) {
          minimo = d;
          punto1 = i;
          punto2 = j;
        }
      }
    }
    System.out.println("Los puntos más cercanos son :");
    System.out.println(p[punto1] + " y " + p[punto2]);
    System.out.println("Con distancia :" + minimo);
  }
}
```

Para mostrar la utilización de los métodos construidos hallaremos la distancia de un punto al punto con el cual una recta tiene la distancia más corta, esto es:

Como se ve en el dibujo lo que se quiere hallar es el punto con el cual una línea trazada desde el punto forman un ángulo recto. Para ésto primero hallamos una recta perpendicular, a la recta que pase por el punto especificado. Luego hallamos la intersección de las dos rectas, y finalmente la distancia entre éstos dos puntos. Como ve no es necesario resolver matemáticamente este problema a fin de codificarlo, lo que se hace es utilizar las funciones disponibles. El resultado es el siguiente:

```
public class PruebaGeo {
/*
 * @author Jorge Teran
 */
  public static void main(String[] args) {
    Punto p4 = new Punto(4,-1);
    Linea l4 = new Linea(3,-4,12);
    Linea l5 = l4.lineaPerpendicularPunto(p4);
    Punto p5 = l5.Interseccion(l4);
    System.out.println("Distancia = "+p5.Distancia(p4));
  }

}
```

9.3. Librerías de Java

En el lenguaje Java se cuentan con funciones trigonométricas en la clase Math, Se presentan las principales en el cuadro 9.1

También se dispone del paquete java.awt.geom que provee una serie de clases que facilitan el desarrollo de programas que tratan con problemas geométricos. Existen clases para puntos, líneas , rectángulos, áreas, y polígonos. Algunos de éstos métodos se muestran en el cuadro 9.2.

El siguiente programa muestra el uso de éste paquete de Java.

Constante PI	Math.PI
Arco coseno	Math.acos(double a)
Arco seno	Math.asin(double a)
Arco tangente	Math.atan(double a)
Seno	Math.sin(double a)
Coseno	Math.cos(double a)
Tangente	Math.tan(double a)
Coseno hiperbólico	Math.cosh(double x)
Seno hiperbólico	Math.sinh(double x)
Valor de la hipotenusa	Math.hypot(double x, double y)
Convertir a grados	toDegrees(double anguloRadianes)
Convertir a radianes	toRadians(double anguloGrados)

Cuadro 9.1: Funciones trigonométricas del Java

Construir un punto	Point2D.Double(double x, double y)
Distancia a otro punto	distance(Point2D p)
Construir una línea	Line2D.Double(Point2D p1, Point2D p2)
Contiene el punto p	contains(Point2D p)
Intersecta con el rectángulo	intersects(Rectangle2D r)
Intersecta la línea	intersectsLine(Line2D l)
Distancia al punto p	ptLineDist(Point2D p)

Cuadro 9.2: Funciones para puntos y líneas

```
import java.awt.geom.*;
/*
 * @author Jorge Teran
 */

public class ClasesJava {

  public static void main(String[] args) {
    Point2D.Double p1 = new Point2D.Double(3, 5);
    Point2D.Double p2 = new Point2D.Double(3, 35);
    System.out.println("p1= " + p1);
    System.out.println("p2= " + p2);
    System.out.println("Distancia de p1 a p2= "
                       + p1.distance(p2));
    Line2D.Double l = new Line2D.Double(p1, p2);
    Point2D.Double p3 = new Point2D.Double(7, 5);
    System.out.println("Distancia al punto P3= "
                      + l.ptLineDist(p3));
    Line2D.Double l2 = new Line2D.Double(p3, p2);
    System.out.println("l intersecta a l2= "
                     + l.intersectsLine(l2));
    System.out.println("l contiene a p2= "
                       + l.contains(p2));

  }
}
```

9.4. Cercos convexos

Dado un conjunto de puntos S se define un cerco convexo, el conjunto convexo más pequeño que contenga a todos los puntos de S.

Cuando graficamos en la pantalla los puntos (x,y) hay que tomar en cuenta que el (0,0) es el extremo superior izquierdo de la pantalla. Consideremos los siguientes puntos:

El cerco convexo se forma con los puntos (86,34), (57,39), (41,98), (110,85). Una propiedad del cerco convexo es que todos los segmentos de recta conformada por los puntos del conjunto están en el interior del cerco convexo. La figura siguiente muestra el cerco y se puede observar que todos los puntos están contenidos por el cerco.

Para hallar cercos convexos existen varios algoritmos desarrollados. Consideremos inicialmente que comparemos un punto con cada uno de los segmentos de recta que se pueden formar y decidir si está al interior del cerco. Este proceso claramente involucra tres ciclos, uno para cada punto y dos para armar los segmentos de recta a probar. Esto genera un algorimo proporcional a $O(n^3)$.

Figura 9.1: Alternativas para comparar un punto con una línea

A fin de implementar la comparación de un punto con respecto a una línea es necesario tomar en cuenta diferentes alternativas. Supongamos que el segmento de recta entre los puntos $p1$ y $p2$ donde el $p1.x = p2.x$, o sea paralelas al eje y. En este caso con referencia al primer punto obtendremos la respuesta verdadera cuando $p1.x < p2.x$. Esto significa que el punto está más a la izquierda de $p1$. La figura 9.1 lo esquematiza.

La figura 9.2 muestra el caso cuando el punto se encuentra en el segmento de recta. Cuando el segmento de recta no es perpendicular al eje y se presentan varios casos de los que solo mostramos algunos en la figura 9.3. Para hallar la posición de un punto con relación a un segmento de recta se ha agregado un método de la clase *Línea* que da *true* si el punto esta más a la izquierda del segmento de recta. El código es el siguiente:

```
public boolean aIzquierda(Punto p) {
  double pendiente=Pendiente();
      if (this.sinPendiente) {
    if (p.x < p1.x)
      return true;
    else {
      if (p.x == p1.x) {
        if (((p.y > p1.y) && (p.y < p2.y))
            || ((p.y > p2.y) && (p.y < p1.y)))
          return true;
        else
```

Figura 9.2: Punto en el segmento de la recta

Figura 9.3: Caso en que el segmento de recta no es perpendicular al eje Y

```
            return false;
        } else
            return false;
      }
    } else {
      double x3 =  (((p.x + pendiente
          * (pendiente * p1.x - p1.y + p.y))
          / (1 + pendiente * pendiente)) );
      double y3 = ((pendiente * (x3 / 1 - p1.x) + p1.y));

      if (pendiente == (double) 0) {
        if ((p.y ) > y3)
          return true;
        else
          return false;
      } else {
        if (pendiente > (double) 0) {
          if (x3 > (p.x ))
            return true;
          else
            return false;
        } else {
          if ((p.x * 1) > x3)
            return true;
          else
            return false;
        }
      }
    }
  }
```

Para construir el código que halle el cerco convexo se ha realizado un programa de ejemplo, que es el siguiente:

```
import java.awt.Graphics;
import java.util.Vector;
```

```
/* Programa oara hallar un
 * cerco convexo
 * @author Jorge Teran
 */

public class Convexo {
  public static void main(String[] args) {
    Vector puntos = new Vector();
    Vector cerco = new Vector();
    int i, j, k;

    puntos.addElement(new Punto(57, 39));
    puntos.addElement(new Punto(41, 98));
    puntos.addElement(new Punto(60, 75));
    puntos.addElement(new Punto(64, 59));
    puntos.addElement(new Punto(110, 85));
    puntos.addElement(new Punto(86, 34));
    puntos.addElement(new Punto(87, 71));

    System.out.println("Punto");
    for (k = 0; k < puntos.size(); k++) {
      System.out.println((Punto) puntos.elementAt(k));
    }
    boolean masIzquierdo, masDerecho;
    for (i = 0; i < puntos.size(); i++) {
      for (j = (i + 1); j < puntos.size(); j++) {
        masIzquierdo = true;
        masDerecho = true;
        Linea temp = new Linea(
            (Punto) puntos.elementAt(i),
            (Punto) puntos.elementAt(j));

        for (k = 0; k < puntos.size(); k++) {
          if ((k != i) && (k != j)) {
            if (temp.aIzquierda(
                          (Punto)puntos.elementAt(k)))
              masIzquierdo = false;
            else
```

```
              masDerecho = false;
            }
          }

          if (masIzquierdo || masDerecho) {
            cerco.addElement(new
            Linea((Punto) puntos.elementAt(i),
                  (Punto) puntos.elementAt(j)));
          }
        }
      }

      System.out.println("Cerco Convexo");
      for (k = 0; k < cerco.size(); k++) {
        System.out.println(((Linea) cerco.elementAt(k)).p1
                 + "," + ((Linea) cerco.elementAt(k)).p2);
      }
    }
}
```

Existen varios algoritmos más eficientes para hallar el cerco convexo de los que se mencionan *Graham*,*Jarvis Quick hull*.

Alejo Hausner del departamento del ciencias de la computación de la universidad de Princeton ha desarrollado unas animaciones que, muestran la ejecución de estos algoritmos.

Se pueden ver en su página web:
http://www.cs.princeton.edu/¬ah/alg_anim/version1/ConvexHull.html

El método de Graham de basa en

- Encontrar el punto más extremo. Se escoge el punto con una coordenada y más grande. Se denomina pivote y se garantiza que está en el cerco convexo.
- Se ordenan los puntos en forma ascendente en función del ángulo que forman con el pivote, terminamos con un polígono con forma de estrella.
- Luego se construye el cerco recorriendo el contorno del polígono.

El método de Jarvis de basa en:

- Se inicializa en algún punto que, se garantiza pertenece al cerco
- Se recorren los puntos buscando cual está más a la izquierda, ese estará en el cerco.

El método de Quick hull de basa en:

- Dado un segmento de línea AB cuyos dos puntos que conocemos están en el cerco convexo.
- Se halla el punto más alejado a este segmento formando un triángulo ABC.
- Los puntos que están en el interior del triángulo no pueden ser del arco convexo. Con los puntos que están afuera del segmento AB se crea un conjunto y con los que están fuera del segmento BC otro. Con estos dos conjuntos se repite el proceso recursivamente.

9.5. Clase Java para polígonos

Las clases de Java para el manejo de polígonos es la clase `Polygon()` que crea un polígono vacío. Los métodos de esta clase son:

Agregar un punto	addPoint(int x, int y)
Verifica si el punto está al interior del polígono	contains(Point p)
Obtiene las coordenadas de un rectángulo que contiene al polígono	getBounds()
Verifica si el polígono intersecta con el rectángulo	intersects(Rectangle2D r)

Cuando construimos un polígono la forma de la figura está dada por el orden en el que ingresamos los puntos. El siguiente código ejemplifica el uso de la clase `Polygon()`.

```
import java.awt.Polygon;
import java.awt.Point;
/*
 * @author Jorge Teran
 */
```

```
public class polígono {
public static void main(String[] args) {
Polygon puntos = new Polygon();

int i, j, k;

puntos.addPoint(57, 39);
puntos.addPoint(41, 98);
puntos.addPoint(60, 75);
puntos.addPoint(64, 59);
puntos.addPoint(110, 85);
puntos.addPoint(86, 34);
puntos.addPoint(87, 71);
Point p= new Point (60,75);
System.out.println(puntos.getBounds());
System.out.println(p);
System.out.println(puntos.contains(p));
}
}
```

Este código inserta los puntos del ejemplo de cerco convexo, produciendo un rectángulo que incluye todos los puntos.

```
java.awt.Rectangle[x=41,y=34,width=69,height=64]
```

Si realizamos la prueba

```
Point p= new Point (60,75);
System.out.println(puntos.contains(p));
```

Veremos que la respuesta es $false$ debido a que la clase polígono no halla un cerco convexo, y este punto no está al interior del polígono. En cambio si creamos un polígono con los puntos (57, 39), (41, 98), (110, 85), (86, 34) obtendremos *true* dado que el punto está en el interior del polígono.

9.6. Cálculo del perímetro y área del polígono.

Para hallar el perímetro de un polígono, es suficiente con hallar la distancia entre dos puntos consecutivos y sumar. Cuando se quiere hallar el área no es tan simple. Consideremos el siguiente polígono

Para poder determinar el área de éste se requiere triangulizar el mismo. Esto es dividir en triángulos y sumar.

El proceso de triangular se inicia numerando los vértices del polígono de 0 a $n-1$. Luego tomamos la línea entre los puntos 0 y $n-1$ como base y buscamos un punto con el cual producen un triángulo que no contenga puntos interiores. De esta forma se divide el polígono en tres partes. Un triángulo y dos subpolígonos uno a la izquierda y otro a la derecha. Si un subpolígono tiene tres lados es un triángulo y no requiere más subdivisiones. Luego este proceso se repite hasta terminar todos los subpolígonos.

Cada triangulización de un polígono de n vértices tiene $n-2$ triángulos y $n-3$ diagonales.

El código para triangular el polígono de ejemplo es el siguiente:

```
import java.awt.Graphics;
import java.util.Vector;
/* Programa para triangular el poligono
 * @author Jorge Teran
```

```
 */

public class T {
  public static void main(String[] args) {
    Vector puntos = new Vector();
    Vector triangulo = new Vector();

    puntos.addElement(new Punto(57, 39));
    puntos.addElement(new Punto(41, 98));
    puntos.addElement(new Punto(60, 75));
    puntos.addElement(new Punto(64, 59));
    puntos.addElement(new Punto(110, 85));
    puntos.addElement(new Punto(86, 34));
    puntos.addElement(new Punto(87, 71));

    Proceso(puntos,triangulo);
      return;
  }
  static void Proceso(Vector puntos, Vector triangulo){
    Vector iPuntos = new Vector();
    Vector dPuntos = new Vector();
    Punto t1,t2,t3;
    int i, j, k;
    i=0; j= puntos.size()-1;
        if (j<2) return;
    if (j==2){
      t1=(Punto) puntos.elementAt(0);
      t2=(Punto) puntos.elementAt(1);
      t3=(Punto) puntos.elementAt(2);
      triangulo.addElement(new Linea(t1,t2));
      triangulo.addElement(new Linea(t1,t3));
      trangulo.addElement(new Linea(t2,t3));
      return;
    }
    k=buscaTriangulo(puntos);
    iPuntos=new Vector();
    iPuntos.add((Punto)puntos.elementAt(0));
    for (i=k;i<puntos.size();i++)
```

```
    iPuntos.add((Punto)puntos.elementAt(i));
  Proceso(iPuntos,triangulo);
  new DibujarPuntos(iPuntos, triangulo);
  dPuntos=new Vector();
  for (i=0;i<=k;i++)
    dPuntos.add((Punto)puntos.elementAt(i));
  Proceso(dPuntos,triangulo);
  new DibujarPuntos(dPuntos, triangulo);
}

static int buscaTriangulo(Vector puntos){
  int i,pi,ult;
  ult=puntos.size()-1;
  for (i = 1;i<puntos.size()-1;i++){
    pi=tienePuntoInterior(puntos,i,ult);
    return pi;
  }
  return 0;
}
static int tienePuntoInterior
                (Vector puntos,int k,int ult){
  Punto t1,t2,t3,t4;
  t1 = (Punto)puntos.elementAt(0);
  t2 = (Punto)puntos.elementAt(ult);
  t3 = (Punto)puntos.elementAt(k);
  Linea l1 = new Linea(t1,t2);
  Linea l2 = new Linea(t1,t3);
  Linea l3 = new Linea(t2,t3);
  int i=-1;
  for (i = 1;i<puntos.size()-1;i++){
    if ((i!=k)&&(i!=ult)){
      t4 = (Punto)puntos.elementAt(i);
      if (l1.aIzquierda(t4)) return i;
      if (l2.aIzquierda(t4)) return i;
      if (l3.aIzquierda(t4)) return i;
    }
  }
  return i;
```

```
  }
}
```

En el libro [Mis97] se muestran diversos algoritmos para triangular polígonos. De los que hay que mencionar el algoritmo de Chazelles que puede triangular polígonos simples en tiempo lineal.

9.7. Ejercicios

Resolver los ejercicios de geometría del Juez de Valladolid.

121, 142, 155, 190, 191, 308, 313, 378, 438, 453, 476, 477, 478, 535, 905, 922, 972, 10112, 10136, 10180, 10195, 10209, 10215, 10221, 10242, 10286, 10297, 10316, 10347, 10351, 10355, 10432, 10451, 10573, 10678, 10897, 10915, 10991, 11152, 11232, 11281, 11314, 11326, 11345, 11373, 11401, 11437, 11479, 11524, 11596.

Resolver los ejercicios de geometría computacional del Juez de Valladolid.

105, 109, 137, 143, 184, 218, 270, 303, 361, 460, 634, 681, 737, 866, 904, 920, 10002, 10005, 10043, 10065, 10078, 10084, 10088, 10117, 10135, 10167, 10169, 10173, 10245, 10263, 10301, 10321, 10348, 10382, 10406, 10553, 10613, 10691, 10697, 10864, 10902, 10927, 11030, 11122, 11156, 11243, 11275, 11277, 11343, 11355, 11378, 11447, 11473, 11562, 11612, 11626, 11696, 11702, 11704, 11707.

Bibliografía

[Cha94] Bernard Chazelle. Computational geometry: a retrospective. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 75–94, New York, NY, USA, 1994. ACM Press.

[CK79] John C. Cherniavsky and SamuelÑ. Kamin. A complete and consistent hoare axiomatics for a simple programming language. *J. ACM*, 26(1):119–128, 1979.

[CSH05] Gary Cornell Cay S. Horstman. *Core Java 2 J2SE 5.0 Volume 1*. Sun Microsystems Press, 2005.

[EH96] Sartaj Sahni Ellis Horowits. *Fundamentals of Algoritms*. Computer Science Press, 1996.

[Fel05] Yishai A. Feldman. Teaching quality object-oriented programming. *J. Educ. Resour. Comput.*, 5(1):1–16, 2005.

[GB96] Paul Bratley Gilles Brassard. *Fundamentals of Algoritms*. Printice Hall, 1996.

[Gri77] Ralph P. Grimaldi. *Matematica Discreta*. Addison Wessley, 1977.

[Hoa83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[KBO+05] Benjamin A. Kuperman, Carla E. Brodley, Hilmi Ozdoganoglu, T.Ñ. Vijaykumar, and Ankit Jalote. Detection and prevention of stack buffer overflow attacks. *Commun. ACM*, 48(11):50–56, 2005.

[Leh88] Charles H. Lehmann. *Geometría Analítica*. Editorial Limusa, 1988.

[MAR03] Steven S. Skiena Miguel A. Revilla. *Programming Challenges*. Springer, 2003.

[McC01] Jeffrey J. McConnell. *Analysis of Algorithms: An Active Learning Approach*. Jones and Bartlett Pub, 2001.

[Mis97] Mishra. Computational real algebraic geometry. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry, CRC Press, 1997*. 1997.

[Red90] Uday S. Reddy. Formal methods in transformational derivation of programs. In *Conference proceedings on Formal methods in software development*, pages 104–114, New York, NY, USA, 1990. ACM Press.

[RLG94] Oren Patashnik Ronald L. Graham, Donald E. Knuth. *Concrete Mathematics*. Addison Wessley, 1994.

[Rol01] Timothy Rolfe. Binomial coefficient recursion: the good, and the bad and ugly. *SIGCSE Bull.*, 33(2):35–36, 2001.

[THCR90] Charles E. Leiserson Thomas H. Cormen and Ronald L. Rivest. *Introduction to Algorithms*. Massachusetts Institute of Technology, 1990.

[Vos01] Tanja Vos. *Métodos Formales Especificación y Verificación*. Umsa, 2001.

Apéndice A

Fundamentos matemáticos

Para el correcto análisis de los programas se requiere recordar algunos conceptos de las matemáticas. Se presentan las fórmulas y métodos que se requieren en una forma simple sin ahondar en desarrollos teóricos y demostraciones.

En el mismo, introducimos las ecuaciones más utilizadas y cuyos desarrollos rigurosos los encontrará en los libros de matemáticas. Algunas referencias a éstos textos están en la bibliografía.

A.1. Recordatorio de logaritmos

Es necesario recordar los siguientes conceptos de logaritmos que son de uso común.

1. $a^b = c$ significa que $\log_a c = b$ donde a se denomina la base de los logaritmos, en el estudio que nos interesa normalmente trabajamos con logaritmos en base 2, pero como se explicó en capítulos anteriores simplemente se utiliza log sin importarnos la base.
2. $\log(ab) = \log(a) + \log(b)$
3. $\log(a/b) = \log(a) - \log(b)$
4. $\log(a^b) = b \log(a)$
5. $\log_b(x) = \log_a(x)/\log_a(b)$
6. $a^{\log b} = b^{\log a}$

A.2. Recordatorio de series

A.2.1. Series simples

Supongamos que $u(n)$ es una función definida para n valores. Si sumamos estos valores obtenemos:

$$s(n) = u(1) + u(2) + u(3)..... + u(n).$$

Es muy común el cambiar la notación a la siguiente forma

$$s_n = u_1 + u_2 + u_3..... + u_n.$$

o simplemente

$$s_n = \sum_{i=1}^{n} u_i.$$

A.2.2. Serie aritmética

Una serie aritmética puede expresarse como una secuencia en la que existe una diferencia constante, llamada razón, entre dos elementos contiguos. La notación matemática para describirla es como sigue:

$$s_n = a, a + d, a + 2d, ..., a + (n - 1)d.$$

La suma de los primeros n términos de una serie aritmética puede expresarse como

$$s_n = \sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2} = an + n(n - 1)\frac{d}{2}$$

donde d es la razón ó sea la diferencia entre dos términos.

A.2.3. Serie geométrica

Una serie geométrica es una secuencia de términos donde los términos tiene la forma

$$a, ar, ar^2, ar^3...., ar^n$$

la suma de ésta serie podemos expresarla como sigue:

$$s_n = \sum_{i=0}^{n} ar^i = \frac{a(1 - r^n)}{(1 - r)}$$

en general la suma de una serie geométrica tiende a un límite solo si r^n también tiende a un límite. De esto podemos decir que una serie geométrica es convergente sí y solo sí $-1 < r < 1$.

A.2.4. Propiedades de la sumatorias

Propiedad distributiva

$$\sum_{i=1}^{n} ci = c \sum_{i=1}^{n} i \tag{A.2.1}$$

Propieadad asociativa

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i \tag{A.2.2}$$

Propiedad conmutativa

$$\sum_{i \in K} a_i = \sum_{p(i) \in K} a_p(i) \tag{A.2.3}$$

A.2.5. Series importantes

$$s_n = \sum_{i=1}^{n} 1 = n \tag{A.2.4}$$

$$s_n = \sum_{i=1}^{n} i = n(n+1)/2 \tag{A.2.5}$$

$$s_n = \sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6 \tag{A.2.6}$$

$$s_n = \sum_{i=1}^{n} i^r = n^{r+1}/r + 1 + p_r(n) \tag{A.2.7}$$

donde r es un entero positivo y $p_r(n)$ es un polinomio de grado r.

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \tag{A.2.8}$$

ésta serie se denomina serie armónica y $\log(n+1) < H_n \leq 1 + \log n$

A.3. Combinatoria básica

Una permutación de n objetos es un arreglo ordenado de los objetos. Por ejemplo si tenemos 3 objetos a, b, c los podemos ordenar de 6 maneras:

a b c	a c b
b a c	b c a
c a b	c b a

El primer elemento se puede escoger de n formas, luego el segundo de $n-1$ maneras, y así sucesivamente por lo que

$$n(n-1)(n-2)...,2,1 = n!$$

Una combinación de r objetos de n objetos es una selección de r objetos sin importar el orden y se escribe:

$$\left(\begin{array}{c} n \\ r \end{array} \right) = \frac{n!}{r!(n-r)!}$$

A.3.1. Fórmulas importantes

$$\left(\begin{array}{c} n \\ r \end{array} \right) = \left(\begin{array}{c} n-1 \\ r-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ r \end{array} \right) \tag{A.3.1}$$

$$(1+x)^n = 1 + \sum_{i=1}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) x^i \tag{A.3.2}$$

$$\sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) = 2^n \tag{A.3.3}$$

$$\sum_{i=0, i\ impar}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) = \sum_{i=0, i\ par}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) \tag{A.3.4}$$

A.4. Probabilidad elemental

La teoría de la probabilidad está asociada a los fenómenos aleatorios, esto significa, los fenómenos cuyo futuro no es predecible con exactitud.

Todas las posibles salidas de un experimento al azar se denomina *espacio muestral*. Por ejemplo cuando lanzamos un dado común existen seis posibles salidas que son los números del 1 al 6. Para este experimento el espacio muestral es $S = \{1, 2, 3, 4, 5, 6\}$. El espacio muestral puede ser finito o infinito, continúo o discreto. Se dice que es discreto cuando el número de puntos de la muestra es finito o si pueden ser etiquetados como 1,2,3,4 y así sucesivamente.

Un evento es una colección de puntos de una muestra que a su vez un subconjunto del espacio muestral. Se dice que un evento ocurre cuando al realizar un experimento aleatorio obtenemos un elemento de un conjunto. Por ejemplo si lanzamos un dado el resultado obtenido es parte del conjunto de números impares. Denominamos al conjunto $\oslash$ el evento imposible y a todo el conjunto muestral al evento universal.

La probabilidad es una medida que asigna a cada evento A del espacio muestral un valor numérico un valor $Pr[A]$ que da la medida de que A pueda ocurrir.

1. Para cada evento de $A \geq 0$
2. $Pr[S] = 1$
3. Si A y B son mutuamente excluyentes vale decir $A \cap B = \oslash$ entonces $Pr[A \cup B] = Pr[A] + Pr[B]$
4. $Pr[\bar{A}] = 1 - Pr[A]$
5. $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$
6. $Pr[A|B] = \frac{Pr[A \cap B]}{Pr[B]}$
7. Dos eventos son independientes si $Pr[A \cap B] = Pr[A]Pr[B]$
8. Para indicar la suma de los eventos de S utilizamos la notación:

$$Pr[X = x] = Pr[A_x] = \sum_{\forall s \varepsilon S} Pr[s]$$

9. La esperanza matemática la representamos como

$$E[X] = \sum_x xp(x) = \sum_{\forall s \varepsilon S} X(s)Pr[s]$$

10. La varianza escribimos como σ_x^2 y la definimos como

$$E[X] = \sum_x x(p(x) - E[X])^2$$

11. La desviación standard se define como la raíz cuadrada de la varianza y se representa por el símbolo σ_x

A.5. Técnicas de demostración

A.5.1. Demostración por contradicción

La prueba por contradicción es conocida como prueba indirecta y consiste en demostrar que la negación de una aseveración nos lleva a una contradicción.

Por ejemplo si deseamos probar que la proposición S es correcta debemos primeramente negar S y luego demostrar que $\neg S$ es falso.

Ejemplo: Demostrar que existe un número infinito de números primos.

Para resolver esto se ve que S está dada por la frase *existe un número infinito de números primos* y la negación de ésta seria *existe un número finito de números primos.* si demostramos que esto es una contradicción entonces la aseveración inicial S es verdadera.

Demostración: Supongamos que existe un número finito de números primos $P = p_1, p_2, ..., p_n$ entonces si encontramos un número primo P_{n+1} habríamos demostrado que el conjunto no contiene a todos los primos por lo cual $\neg S$ es falso. Sea $P_{n+1} = p_1 * p_2....p_n + 1$ claramente se ve que el resultado es un número primo, por lo tanto se demuestra que el conjunto de números primos no es finito y queda demostrada nuestra hipótesis original *el conjunto de números primos es infinito.*

A.5.2. Demostración por inducción

Existen dos técnicas que normalmente se utilizan que son la deducción y la inducción. La deducción consiste en ir de un caso general a un caso particular.

En cambio la inducción matemática trata de establecer una ley general en base a instancias particulares. Este razonamiento puede llevar a resultados erróneos si no se plantea adecuadamente. Sin embargo la deducción siempre lleva a resultados correctos.

El principio de inducción matemática se basa, primeramente demostrar la validez de nuestras suposiciones para valores de $0, 1, 2, ..$ y luego de que, nos hacemos alguna idea de como se comporta suponemos que, para un valor arbitrario n es correcto. En base de ésta suposición demostramos que para $n - 1$ es correcto con lo cual se demuestra nuestra suposición original.

Apéndice B

El Juez de Valladolid

El Juez de Valladolid es un programa de computación implementado por la Universidad de Valladolid para revisar que los problemas propuestos son resueltos adecuadamente.

La importancia del Juez virtual en el ámbito educativo es reconocida mundialmente y permite que los estudiantes puedan obtener una respuesta rápida sobre el problema resuelto. El sistema evalúa los problemas por el sistema de casos de prueba. Esto significa que el sistema tiene una serie de casos de prueba que el programa deberá procesar y luego las respuestas serán comparadas con las que tiene el Juez para determinar la correctitud del programa.

La dirección web del juez es $http://uva.onlinejudge.org/$. En este sitio es donde se envían los problemas para su evaluación.

Existen más de 2500 problemas de las diferentes áreas de la informática, tales como, Estructura de datos, Geometría Computacional, Combinatoria, Grafos, Programación Dinámica, etc.

B.1. Como obtener una cuenta

Para enviar problemas al Juez de Valladolid es necesario tener una cuenta con usuario y contraseña, que se la puede obtener gratuitamente completando el registro en la página web. Cuando se ingresa a la página se obtiene la siguiente pantalla:

Ahora debe escoger la opcion *Register* y obtener la patanlla de registro donde debe llenar los datos solicitados como se muestra:

Hay que tomar en cuenta algunos aspectos al momento de realizar el registro. La dirección de correo debe estar correctamente escrita dado que le llegara un correo solicitando la confirmación para activar su cuenta. El nombre de usuario y la contraseña es la que utilizará para acceder a su cuenta.

Como esta cuenta no proviene del antiguo sistema, es una cuenta nueva, en el campo *Online Judge ID* debe poner un 0.

El campo *Results email* se utiliza para obtener las respuestas del Juez en su cuenta de correo.

Una vez que ingrese al sistema la pantalla que obtendrá es la siguiente:

Una vez en el sistema seleccione *Browse problems* y obtendrá una lista de problemas catalogadas como: problemas del juez, problemas de concursos, problemas del libro *Programming Challenges*, de torneos mundiales y otros.

Escogemos los problemas del juez, luego volumen 1, y el primer ejercicio el $3n + 1$ y tenemos:

Los enunciados de problemas nos presentan cuatro partes

1. Antecedentes.- Que describen y motivan el problema.

2. Datos de entrada.- Aquí se describe el formato de los datos que el programa deberá procesar.

3. Datos de salida.- Que muestran como deben imprimirse los resultados.

4. Ejemplo de datos de entrada.- Se presentan algunos ejemplos de como los datos de entrada vendrán presentados. Obviamente en el sistema del Juez hay muchísimos más datos.

5. Ejemplo de datos de salida.- Se muestra como para los datos de entrada del ejemplo será la respuesta que debe imprimir el programa.

El programa que se realice debe leer los datos de entrada del teclado *(standard input)* y escribir la salida en pantalla *standard output*.

Las restricciones del tiempo de ejecución en algunos problemas están dadas al comienzo del enunciado. En cuanto a la memoria también esta restringida a 64mb.

B.2. Como enviar un problema

Una vez que ha resuelto y probado su programa puede enviarlo al Juez para probar si está correcto escogiendo *submit*.

Para enviar su problema primero debe escoger el lenguaje de programación utilizado. El Juez acepta solo C, C++, Java, y Pascal.

Los lenguajes C, y C++ son los compiladores de GNU versiones 4.1.2 que son gratuitos y pueden descargarse de la página web *http://gcc.gnu.org/*.

El lenguaje Java es la versión 6 de Sun Mricrosystems que puede descargarse de la página *http://www.java.com/es/download/*.

El lenguaje Pascal es del Free Pascal Compiler versión 2.04 que también es gratuita y puede descargarse de *http://www.freepascal.org/download.html*.

Esto es muy importante dado que de un proveedor a otro pueden existir diferencias en la sintaxis de los programas.

Una vez escogido el lenguaje tienen dos opciones. La primera copiar el código en el cuadro de texto o la más recomendable, la de subir el código fuente con la opción examinar y luego seleccionado submit.

Un detalle muy importante es que el programa debe llamarse $Main$.

Para conocer si su programa resuelve adecuadamente el problema debe seleccionar la opción *My Submissions* y tendrá la lista de los ejercicios enviados y si fueron resueltos correctamente.

Las posibles respuestas que obtendrá son:

- Accepted.- El ejercicio fue resuelto correctamente.

- Runtime error.- Error en tiempo de ejecución. Puede ser, por ejemplo, un índice fuera de rango, división por cero, etc.
- Compilation error. Error de compilación.
- Wrong answer.- La salida del programa es incorrecta.
- Time limit exceded.- El tiempo de ejecución excede al permitido. Puede darse si selecciona un algoritmo más lento o su programa es poco eficiente.
- Presentation errror.- El formato de salida es incorrecto, sin embargo la respuesta es correcta.

B.3. Problema de ejemplo

Para ejemplificar diferentes errores y mostrar la lectura de datos resolveremos el problema 100, $3n + 1$- Veamos el enunciado:

El problema de 3n + 1

Consideremos el siguiente algoritmo para generar una secuencia de números. Comenzando con un entero n: si n es par, se divide por 2; si n es impar, se multiplica por 3 y se suma 1. Este proceso se debe repetir para cada nuevo valor de n, finalizando cuando $n = 1$. Por ejemplo, para n = 22 se genera la siguiente secuencia de números:

```
22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Se conjetura (aunque no está demostrado) que este algoritmo termina en $n = 1$ para cualquier entero n. Dicha conjetura se cumple, al menos, para cualquier entero hasta 1.000.000.

Para una entrada n, la longitud de ciclo de n es la cantidad de números generados hasta e incluyendo, el 1. En el ejemplo anterior, la longitud de ciclo de 22 es 16. Dados dos números cualesquiera, i y j, se debe determinar la máxima longitud de ciclo correspondiente a un número comprendido entre i y j, incluyendo ambos extremos.

Entrada La entrada consta de una serie de parejas de enteros, i y j, habiendo una pareja por línea. Todos los enteros serán menores de 1.000.000 y mayores de 0.

Salida Para cada pareja de enteros i y j, escribir i y j en el mismo orden en el que aparecen en la entrada, seguidos de la longitud de ciclo máxima para los enteros comprendidos entre i y j, ambos incluidos. Los tres números deben estar separados entre sí por un espacio, estando los tres en la misma línea y utilizando una nueva línea en la salida por cada línea que aparece en la entrada.

Ejemplo de salida	Ejemplo de entrada
1 10	1 10 20
100 200	100 200 125
201 210	201 210 89
900 1000	900 1000 174

Analizando el enunciado se ve que lo que se pide primero, es leer dos números enteros del teclado. En Java esto se resuelve, utilizando la clase *Scanner*.

```
Scanner lee = new Scanner(System.in);
       i = lee.nextInt();
       j = lee.nextInt();
```

Para saber si hay más valores que leer se utiliza el método *hasNext()*.

Luego para un n dado, cuantas iteraciones realiza el algoritmo hasta que n tome el valor de 1.

```
Para un entero n: si n es par, se divide por 2;
si n es impar, se multiplica por 3 y se suma 1
```

Aquí debemos recordar dos aspectos importantes. El primero es como determinar si un número es par o impar. Comúnmente en función de la definición probamos si la división por 2 da como resto cero. Esto equivale a escribir $n\,\%2 == 0$. Analizando esta instrucción vemos que implica una división, una multiplicación y una resta.

Si vemos el problema en números binarios todos los números pares terminan en un dígito cero. Por lo que podemos simplificar el proceso a preguntar si el ultimo dígito binario es cero. La instrucción $((n\&1)! = 0)$ toma el número uno y después de una operación *and* en binario. Si el resultado es 0 es par, caso contrario es impar. Como las computadoras están hechas para trabajar en binario comprenderá que esto es mucho mas rápido.

Es estrictamente necesario para evitar el error límite de tiempo excedido.

Luego se ve necesario hacer una división por 2. Como conocemos las divisiones toman un tiempo excesivo por que es aconsejable evitarlas. Cuando multiplicamos por 10 en números decimales agregamos un cero a la derecha. En números binarios ocurre lo mismo, cuando multiplicamos por la base en, es decir, 2 agregamos un cero a la derecha.

Al dividir ocurre lo mismo se quita el dígito de derecha. En Java se codifica utilizando las instrucciones << y >> que recorren los bits hacia la derecha y a la izquierda respectivamente.

Entonces para dividir por 2 recorremos todos los bits una posición a la derecha con la instrucción:

```
n >>= 1;
```

Otra vez esto es necesario para entrar dentro de los tiempos de ejecución permitidos.

Finalmente el programa pide imprimir el resultado que se realiza con las instrucciones:

```
System.out.println(i + " " + j + " " + maximo);
```

Hay que considerar que la salida debe ser idéntica a la solicitado. Vea que **no** hemos escrito frases, tales como, *el resultado es*, *el máximo es* porque los mismos harían activar el mensaje respuesta incorrecta.

El programa que resuelve el problema planteado es:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        int i, j, desde, hasta, num, n, k, maximo;
        Scanner lee = new Scanner(System.in);
        while (lee.hasNext()) {
            i = lee.nextInt();
            j = lee.nextInt();
            if (i < j) {
                desde = i;
                hasta = j;
            } else {
                desde = j;
                hasta = i;
```

```
            }
        for (maximo = -1, num = desde; num <= hasta; num++) {
                n = num;
                k = 1;
                while (n > 1) {
                    if ((n & 1) != 0)
                        n = 3 * n + 1;
                    else
                        n >>= 1;
                    k++;
                }
                if (k > maximo)
                    maximo = k;
            }
            System.out.println(i + " " + j + " "
                    + maximo);

        }
    }
}
```

B.4. Ayuda para corregir errores

El sitio http://uvatoolkit.com/ es un sitio construido por la Universidad de Valladolid, orientado a proveer ayudas en la solución de ejercicios.

Cuando selecciona un ejercicio en la página obtiene las ayudas necesarias, para que, con su datos de prueba pueda generar resultados y probar sus programas. Si escogemos el problema 100 que acabamos de realizar veremos la pantalla siguiente:

En estos campos podemos colocar los datos de prueba que queremos y ver cual es el resultado. De esta forma podemos comparar esta salida con la de nuestro programa.

B.5. Ejercicios

Para practicar la lista de ejercicios del Juez de Valladolid que se presenta, no requiere utilizar técnicas específicas como las descritas en el libro.

100, 101, 118, 119, 127, 130, 133, 139, 141, 144, 145, 151, 154, 159, 161, 162, 170, 183, 187, 188, 201, 227, 232, 253, 272, 278, 297, 300, 305, 333, 337, 344, 353, 362, 371, 384, 392, 394, 401, 402, 409, 413, 414, 422, 440, 444, 445, 448, 455, 457, 458, 466, 474, 483, 484, 486, 488, 489, 490, 492, 494, 499, 537, 541, 575, 576, 579, 585, 587, 591, 594, 603, 608, 621, 628, 637, 640, 641, 644, 661, 694, 700, 706, 739, 740, 834, 850, 895, 957, 10010, 10018, 10019, 10033, 10035, 10037, 10038, 10045, 10055, 10082, 10101, 10114, 10116, 10126, 10134, 10137, 10141, 10142, 10145, 10146, 10156, 10172, 10188, 10189, 10190, 10191, 10196, 10197, 10205, 10222, 10227, 10252, 10258, 10260, 10267, 10279, 10281, 10282, 10293, 10295, 10298, 10300, 10315, 10324, 10346, 10361, 10370, 10374, 10377, 10409, 10415, 10424, 10427, 10443, 10469, 10500, 10502, 10530, 10550, 10554, 10611, 10656, 10703, 10730, 10783, 10800, 10812, 10878, 10901, 10903,

10908, 10919, 10921, 10945, 10963, 11040, 11048, 11051, 11057, 11063, 11150, 11172, 11192, 11203, 11204, 11219, 11220, 11221, 11222, 11223, 11225, 11233, 11241, 11244, 11247, 11278, 11279, 11286, 11308, 11309, 11313, 11332, 11340, 11348, 11349, 11356, 11357, 11360, 11364, 11385, 11403, 11452, 11470, 11480, 11482, 11483, 11494, 11496, 11497, 11498, 11499, 11508, 11511, 11512, 11530, 11541, 11556, 11559, 11576, 11577, 11581, 11588, 11608, 11623, 11638, 11661, 11677, 11683, 11687, 11697, 11713, 11716, 11717, 11727, 11734, 11743.

Índice alfabético

árbol, 173
  ejemplo, 173

, 221

afirmaciones, 98
  assert, 99
  Java, 99
algorítmica elemental, 1
Algoritmo, 1
análisis amortizado, 17
apriori, 3
ArrayList
  métodos, 160

búsqueda, 103
  binaria, 105, 107
  eficiencia, 103
  secuencial, 103, 104
    eficiencia, 104
Backtracking, 179

cadenas, 142
Carmichael, 53
Cercos convexos, 202
  ejemplo, 206
  métodos, 208
clasificación, 103, 114
  arrays, 115
  burbuja, 120
    eficiencia, 121
  comparación, 131
  compareTo, 117
  inserción, 121
    invariante, 121
  Java, 115
  lineal, 128
    eficiencia, 129
  pivote, 125
  postcondición, 120
  precondición, 120
  rápida, 125
    eficiencia, 127
  representación, 119
  selección, 123
    eficiencia, 124
    invariante, 124
  sort, 115
coeficientes binomiales, 143
  aplicaciones, 143
cola prioridad, 175
  ejemplo, 175
combinatoria, 141
  contar, 141
complejidad, 11
congruencias, 46
  ecuaciones, 46, 47
  inverso, 46
  potencias, 46
  propiedades, 46
  reflexiva, 46

simétrica, 46
suma, 46
transitiva, 46
conjuntos, 168
ejemplo, 170
Construir permutaciones, 185
en desorden, 187
en orden, 188

demostración por contradicción, 224
demostración por Inducción, 224
desviación standard, 224
divide y vencerás, 22
divisibilidad, 39
divisor común, 39

eficiencia, 1–3, 7, 11, 103, 127, 131, 134
empírico, 2
Eratostenes, 48, 100
errores de programación, 68
ejemplo, 68, 70
ejemplo C, 69
espacio muestral, 223
especificación, 72
especificar, 109
estructuras de datos, 157
estructuras recursivas, 14
estructuras secuenciales, 14
Euclides, 40, 80
Euclides complejidad, 40
Euclides extendido, 42

factores primos, 50
factorización, 50
Fermat, 53
Fibonacci, 24, 88, 145

geométrica, 221
geometría, 193, 194
Java, 200
línea, 196
punto, 194
grafo, 179
grafos
ejemplo, 180
recorrido, 179

ineficiencia, 121
instancia, 2, 3, 6, 7, 10, 11
invariante, 73, 74, 76–86, 91
aplicaciones, 81

listas enlazadas, 165
ejemplo, 167
métodos, 166
logaritmos, 12, 219

máximo común divisor, 38–40
método iterativo, 20
método teórico, 3
mínimo común múltiplo, 44
map, 171
ejemplo, 172
master, 22
Miller - Rabin, 53
modular, 44, 45

número Euleriano, 147
invariante, 149
triángulo, 148
número Stirling, 150
números Catalanes, 146
aplicaciones, 147
números grandes, 54
números primos, 48, 50
generación, 48
notación asintótica, 3, 13

notación omega, 13
notación teta, 13

O grande
    igualdad logaritmos, 12
    límites, 12
    propiedades, 12
    transitiva, 12
operaciones elementales, 6
orden de, 11

permutación, 142, 222
pila, 162
    ejemplo Java, 162
    lista LIFO, 162
polígono, 193
    área, 194
    búsqueda, 193
    convexo, 193
    intersección, 193
    partición, 193
polígonos
    Java, 209
Post condición, 73, 74, 78, 80–83, 85–87
post condición, 97
precondición, 72, 73, 80, 81, 97
probabilidad, 223
probable primo, 55
programa certificado, 108, 109
programa probado, 108
prueba de primalidad, 52
prueba de programas, 67
    caja blanca, 67
    caja negra, 67
    verificación, 67
prueba dinámica, 98
prueba estática, 97
    proceso, 97
prueba exaustiva, 107

recurrencias, 18
recurrencias lineales, 23
regla de la suma, 142
regla del producto, 141
regla exclusión, 142
regla inclusión, 142
representación gráfica, 110
RSA, 56

Serie aritmética, 220
Serie geométrica, 220
Series Simples, 220
Subconjuntos, 183
    ejemplo, 184
subconjuntos, 142
substitución, 18
sumatoria
    propiedades, 221

teórica, 3
teoría de números, 35
tiempo proceso operaciones, 35
tipos de algoritmos, 3
triángulo de Pascal, 143
    invariante, 144
triangulación, 193
triangular, 211

variables Java, 35
Vector
    métodos, 159