ALGORITMOS PROBABILÍSTICOS

MAYERLIN ASTUDILLO ASTUDILLO

UNIVERSIDAD DEL CAUCA
FACULTAD DE CIENCIAS NATURALES, EXACTAS Y DE LA EDUCACIÓN
DEPARTAMENTO DE MATEMÁTICAS
POPAYÁN
2004

ALGORITMOS PROBABILÍSTICOS

MAYERLIN ASTUDILLO ASTUDILLO

TRABAJO DE GRADO

En la modalidad de seminario de grado presentado como requisito parcial para optar al título de matemática

Director

Mg. MAURICIO MACA CHAGÜENDO

UNIVERSIDAD DEL CAUCA

FACULTAD DE CIENCIAS NATURALES, EXACTAS Y DE LA EDUCACIÓN

DEPARTAMENTO DE MATEMÁTICAS

POPAYÁN

2004

Nota de aceptación

_____

_____

_____

Director

_____

Magister Mauricio Maca Chagüendo

Comité evaludor

_____

Especialista Diego Correa Cuene

_____

Especialista Hebert Vivas

Fecha de sustentación: Popayán, 13 de febrero de 2004

A mi madre y a mis hermanos Doralys, Alexander y Edwin por su constante apoyo, por su amor incondicional, comprensión y paciencia.

A la memoria de Mi papá Lorenzo Y de mi hermano Wilson por haberle dado siempre lo mejor a mi familia.

AGRADECIMIENTOS

Agradezco de todo corazón a:

Mauricio Maca Chagüendo, profesor de matemáticas, jefe del departamento Matemáticas y director del seminario de grado, por su valiosa colabaración y orientación.

Diego Correa Cuene, profesor de Matemáticas y miembro del comité evaluador por su interés y oportunas sugerencias.

Hebert Vivas, profesor de Matemáticas y miembro del comité evaluador por su interés y oportunas sugerencias.

Luis Eduardo Montoya, profesor de Matemáticas y decano de la facultad de Educación por su motivación y ayuda en la realización de este trabajo.

Fredy Amaya, profesor de matemáticas por su motivación y ayuda en la realización de este trabajo.

Yilton Riascos, profesor de Matemáticas por su colaboración y apoyo.

Emérita Astudillo, por su apoyo permanente y su constante motivación en el transcurso de mis estudios.

Doralys, Alexander y Edwin Astudillo Astudillo, por su motivación y apoyo en el transcurso de mi carrera.

Luis Eduardo Egas Pacheco, por su amor, constante apoyo y motivación.

Yudy Marcela Bolaños, profesora de Matemáticas, por su ayuda en la realización de este trabajo.

Maritza Herrera Florez, profesora de Matemáticas, por su ayuda en la realización de este trabajo.

Edward Olmedo Pacheco Castillo, por su valiosa colaboración, apoyo incondicional y oportunos consejos.

A mis compañeros de carrera y amigos, por su constante apoyo.

A la universidad del Cauca, a todos los profesores del departamento de Matemáticas y demás personas que de alguna manera colaboraron con la realización del seminario de grado.

CONTENIDO

INTRODUCCIÓN **1**

1. GENERALIDADES Y EJEMPLOS **3**

1.1. TIEMPO ESPERADO Y TIEMPO PROMEDIO 4

1.2. NÚMEROS SEUDOALEATORIOS 5

1.3. EJEMPLOS 6

1.3.1. Random walks(recorrido aleatorio) 6

1.3.2. Identidades polinomiales 12

1.3.3. Verificando Matching en grafos 15

2. CLASES DE ALGORITMOS PROBABILÍSTICOS **20**

2.1. ALGORITMOS DE MONTE CARLO 21

2.1.1. Verificación de la multiplicación de matrices 21

2.1.2. Verificador de composición 27

2.2. ALGORITMOS DE LAS VEGAS 42

2.2.1. Ordenación probabilísta 44

2.2.2. Factorización de enteros grandes 53

3. MÁQUINAS DE TURING PROBABILÍSTICAS Y SUS CLASES DE COMPLEJIDAD **65**

3.1. MÁQUINAS DE TURING PROBABILÍSTICAS 66

3.1.1. PP-máquinas 74

3.1.2. BPP-máquinas 81

3.1.3. R-máquinas 86

3.1.4. ZPP-máquinas 89

3.2. CLASES DE COMPLEJIDAD PROBABILÍSTICA 92
3.3. LENGUAJES PP-COMPLETOS 102

4. CONCLUSIONES **109**

BIBLIOGRAFÍA **111**

LISTA DE TABLAS

1.1. permutaciones que corresponden a ciclos pares del grafo G de la figura 1.2 19
2.2. Números enteros n y la frecuencia de sus testigos de composición 29
2.3. Paridades para la terna $\langle r_1, r_2, r_3 \rangle$ 40
2.4. Tiempo empleado para la factorización de enteros de diferente número de dígitos 56
3.5. Función de transición de la MTD 69

LISTA DE FIGURAS

1.1. Grafo G con conjunto de vértices $\{1, 2, 3, 4, 5, 6\}$ 16
1.2. Grafo G con conjunto de vértices {1, 2, 3, 4} 19
2.3. Procedimiento partición para el vector A del ejemplo 2.2.1 47
2.4. Dos niveles de el árbol de recursión para el procedimiento Partición 48
3.5. Una computación de MTD dada por la función de transición representada en la tabla 3.5, para la entrada $x = 111$ 70
3.6. Primeros tres niveles del árbol de computación asociado una $MTN(x)$ . . . 71
3.7. Primeros cuatro niveles del árbol de computación asociado una MTP(x) . . 73
3.8. Una PP-máquina 75
3.9. Una BPP-máquina 82
3.10. Una R-máquina 88
3.11. Una ZPP-máquina 90
3.12. Relaciones entre las diferentes clases de complejidad 97

RESUMEN

Este documento contiene el informe del seminario de grado títulado "Algoritmos probabilísticos" desarrollado en el marco de las actividades del grupo de estudio y desarrollo investigativo en matemática aplicada, en la línea de matemática computacional. Inicialmente se presentan algunos ejemplos de este tipo de algoritmos para después mostrar las diferentes clases de algoritmos probabilísticos que se conocen: *Algoritmos de las Vegas* y *algoritmos de Monte Carlo*; se ilustran algunos ejemplos de cada una de estas clases y se analiza la probabilidad de que estos algoritmos siempre encuentren una respuesta correcta, así como la probabilidad de error y el tiempo de complejidad de tales algoritmos. Finalmente, se formaliza el concepto de algoritmo probabilístico, utilizando para ello el modelo de computación teórico conocido como máquina de Turing no determinística con algunas características especiales, algunos de los temas desarrollados en el seminario de grado se han complementado con ejemplos, tablas y gráficas para una mayor facilidad en su comprensión.

INTRODUCCIÓN

El primer ejemplo de algoritmo probabilístico que se conoce es el algoritmo de Berlekamp (1970) utilizado para factorizar un polinomio f sobre el cuerpo $GF(p)$ de p elementos. Este algoritmo corre en tiempo polinomial sobre el grado de f y con probabilidad de al menos un medio de encontrar una factorización correcta; dado que el algoritmo se puede repetir cualquier número de veces y la probabilidad de fallo es siempre independiente, en la práctica el algoritmo siempre factoriza f en tiempo polinomial.

En 1977, Solovay y Strassen muestran una nueva clase de algoritmo para verificar si un número dado es primo. Tal algoritmo escoge aleatoriamente un testigo de la no primalidad de dicho número, esto es un testigo de la composición del número dado; ellos argumentan que si el número no es primo, con alta probabilidad un testigo de la composición de dicho número será encontrado. Históricamente se dice que estos dos ejemplos fueron los que iniciaron el análisis de esta nueva clase de algoritmos, los cuales han atraído el interés de los investigadores de manera creciente en los últimos 20 años y actualmente son un tema central de investigación en Ciencias de la Computación, Investigación Operativa, y Matemática Discreta. Por otro lado, los algoritmos probabilísticos ya han mostrado su utilidad en una gran variedad de aplicaciones en los campos de estructuras de datos, programación lineal y problemas de grafos.

Existen una infinidad de problemas que pueden resolverse de forma más eficiente usando los llamados algoritmos probabilísticos, ya que cuando un algoritmo se enfrenta a una decisión, a veces es preferible seguir un curso aleatorio, en lugar de invertir tiempo en determinar cual alternativa es la mejor para resolver el problema.

Los algoritmos probabilísticos se han clasificado en *algoritmos de Monte Carlo* y *algoritmos de las Vegas*. Los algoritmos de Monte Carlo retornan una respuesta exacta con

una probabilidad muy alta de ser la correcta, sea cual sea el caso considerado, aunque ocasionalmente cometen algún error sin dar aviso; la característica más importante de los algoritmos de Monte Carlo es que suele ser posible reducir arbitrariamente la probabilidad de error con un ligero aumento en el tiempo de ejecución. Por el contrario, los algoritmos de las Vegas nunca devuelven respuestas equivocadas, estos algoritmos toman decisiones probabilísticas como ayuda para guiarse más rápidamente hacia una solución correcta y son confiables porque en lugar de retornar respuestas incorrectas retornan un "no sé" si no están seguros.

Esta nueva clase de algoritmos sugiere revisar la noción de "eficiencia computacional"que hasta el momento se tenía. ¿Podrían considerarse iguales, la clase de problemas computacionalmente eficientes con la clase de problemas solubles polinomialmente por algoritmos probabilísticos?. Para responder esta pregunta se desarrolla una nueva área en la teoría de la complejidad computacional, llamada complejidad probabilística la cual ayuda a entender el poder de la computación probabilística y a formalizar el concepto de algoritmos probabilísticos, utilizando para ello el modelo básico de computación máquina de Turing no determinística.

Como se ha dicho, el estudio de los algoritmos probabilísticos es reciente. La investigación y el conocimiento acerca del poder de la aleatoriedad está todavía en sus comienzos. El presente documento pretende proporcionar un texto que recopile en forma amplia y detallada el estudio de esta nueva clase de algoritmos. Por otra parte, se espera que este trabajo contribuya a fortalecer académico del grupo de estudio y desarrollo investigativo en matemática aplicada, en la línea de matemática computacional, el cual viene trabajando en algoritmos de aproximación para problemas NP-duros. También se espera que el texto sirva de material de consulta y apoyo para estudiantes que deseen iniciarse en el interesante campo de la matemática computacional y quieran continuar sus estudios en esta área, ya que muchas cuestiones interesantes y problemas básicos para esta nueva clase de algoritmos siguen abiertos, siendo un atractivo campo de investigación.

1. GENERALIDADES Y EJEMPLOS

Hay muchos problemas computacionales para los cuales el algoritmo más apropiado está basado en la aleatoriedad, que es la posibilidad de algunos algoritmos de realizar una elección al azar. La teoría de la probabilidad tiene diferentes aplicaciones en el ánalisis de algoritmos, para las entradas por ejemplo, en lugar de considerar la complejidad en el peor de los casos de un algoritmo (como se ha considerado hasta ahora), las técnicas probabilísticas pueden ser usadas para evaluar la complejidad en el caso promedio del mismo algoritmo con respecto a una distribución de probabilidad dada.

Intuitivamente, un **algoritmo probabilístico** es un procedimiento computacional que hace **elecciones aleatorias** de lo que hay que hacer en un momento dado, o de otra manera es un algoritmo determinístico que hace escogencias aleatorias en el curso de sus ejecuciones.

El término **aleatorio** no quiere decir no determinístico (arbitrario); por el contrario, se refiere a valores seleccionados de tal manera que la probabilidad de seleccionar cada uno de ellos sea conocida y esté controlada.

Una instrucción tal como "seleccionar un número entre 1 y 6" si no se dan más datos no es admisible por este tipo de algoritmos, sin embargo sería aceptable decir "seleccionar un número entre 1 y 6 de tal manera que todos los valores tengan la misma probabilidad de ser seleccionados". En un computador podemos implementar este tipo de algoritmos usando un generador de números seudoaleatorios, el cual será descrito más adelante en la sección 1.2. La característica principal de estos algoritmos es que un mismo algoritmo se

puede comportar en formas distintas cuando es aplicado dos veces a una misma entrada. Su tiempo esperado, incluso el resultado obtenido, pueden variar considerablemente entre ejecuciones consecutivas [BB97].

Dos posibles aplicaciones de este tipo de algoritmos son:

1. Ver estos algoritmos resolviendo problemas en **P** [1], los cuales son más eficientes (o mucho más simples) que los algoritmos deterministas que se conocen previamente. Un ejemplo de esta aplicación es, el algoritmo que resuelve el problema del matching perfecto. Sección 1.3.3, página 16.

2. Los algoritmos probabilísticos tiempo polinomial, pueden también ser aplicables para resolver problemas que no pertenecen a la clase **P**, es decir problemas para los cuales no se conoce algoritmo determinístico tiempo polinomial que los resuelva. El problema de identidades polinomiales, sección 1.3.2, página 12, es un ejemplo de esta aplicación de los algoritmos probabilísticos en este tipo de problemas.

Para familiarizarse con los algoritmos probabilísticos y su análisis, se presentan en esta sección tres ejemplos: *identidades polinomiales, ramdon walk (camino aleatorio), y verificando matching en grafos*. Se muestra como un algoritmo probabilístico con error de probabilidad acotado, puede ser usado para obtener una respuesta correcta con probabilidad tan alta como sea posible, esto es, hacer el error de probabilidad de tales algoritmos arbitrariamente pequeño simplemente ejecutando el algoritmo un número limitado de veces sobre la misma entrada, con escogencias aleatorias independientes.

1.1. TIEMPO ESPERADO Y TIEMPO PROMEDIO

Para un buen ánalisis de los algoritmos probabilísticos es importante hacer una distinción entre las palabras *promedio* y *esperado*.

El ***tiempo promedio*** de un algoritmo determinístico, se refiere al tiempo promedio requerido por el algoritmo cuando se consideran igualmente probables todas las entradas

[1] Un problema pertenece a la clase P, si existe un algoritmo determinístico tiempo polinomial que lo resuelve.

de un tamaño dado; este suele ser un requisito poco realista, sobre todo cuando se utiliza un algoritmo como procedimiento interno de otro algoritmo más complejo.

Existen ciertos algoritmos para los cuales no es necesario que las entradas sean igualmente probables, es decir el comportamiento es independiente de los casos concretos que haya que resolver (Algoritmos de las Vegas). **El tiempo esperado de un algoritmo probabilístico** se define para cada entrada y se refiere al tiempo promedio que se necesita para resolver dicha entrada una y otra vez.

Es entonces significativo hablar del *tiempo promedio esperado* y *Tiempo promedio esperado en el peor de los casos de un algoritmo probabilítico*. Este último se refiere al tiempo esperado que requiere el peor caso posible de un tamaño dado y no al tiempo requerido si lamentablemente se toman las peores elecciones probabilísticas. Los algoritmos de las Vegas pueden ser más eficientes que los algoritmos deterministas pero sólo con respecto al tiempo esperado, ya que existen algoritmos deterministas que dan soluciones en el caso promedio en tiempo polinomial y en el peor de los casos en tiempo cuadŕatico y sin embargo, para estos algoritmos existen algoritmos probabilísticos cuyo tiempo esperado en el peor de los casos es lineal. Su rendimiento esperado es considerablemente mejor que el del algoritmo determinista en todos y cada uno de los casos; el único problema de estos algoritmos es la posibilidad de que en alguna ocasión requiera un tiempo excesivo independientemente del caso en cuestión. Algunos ejemplos de este tipo de algoritmos son: el algoritmo que encuentra la mediana de un vector y el algoritmo quicksort. La ventaja de hacer probabilístico un algoritmo determinista que tenga buen comportamiento promedio, a pesar de un peor caso nada eficiente, es hacerlo menos vulnerable a una distribución de probabilidad inesperada de los casos (entradas) que intente resolver.

1.2. NÚMEROS SEUDOALEATORIOS

Para continuar con el desarrollo de la teoría de los algoritmos probabilísticos, se tendrá en cuenta el comportamiento de la función Random (m, n). Esta función toma valores uniformemente distribuidos entre m y n, y devuelve un número real x seleccionado

aleatoriamente en el intervalo $[m, n]$. Llamadas sucesivas del generador devuelven valores independientes de x. Es claro que en la práctica tales generadores de números aleatorios no existen, en la mayoría de los casos se emplean generadores seudoaleatorios.

Los ***generadores seudoaleatorios*** son procedimientos deterministas capaces de generar largas sucesiones de valores que parecen tener la propiedad de una sucesión aleatoria. Para dar inicio a una sucesión se dá un valor inicial llamado semilla, una misma semilla devuelve una misma sucesión. Empleando un buen generador seudoaleatorio, se espera que sean válidos los resultados obtenidos con respecto a la eficiencia y la confiabilidad de los diferentes algoritmos probabilísticos.

1.3. EJEMPLOS

1.3.1. Random walks(recorrido aleatorio)

El primer ejemplo de algoritmos probabilísticos que se verá es un problema en lógica, el problema de satisfabilidad SAT[2]. Antes de definir el problema SAT se harán las siguientes aclaraciones.

- Una colección C de cláusulas sobre un conjunto $U = \{x_1, x_2, \ldots, x_n\}$ de variables, puede ser vista como una fórmula Φ en forma normal conjuntiva, es decir, como una conjunción de disyunciones.
- Una asignación de verdad para U es una función $T : U \to \{v, f\}$ tal que:
  - Si $T(x) = v$ entonces x es verdadera.
  - Si $T(x) = f$ entonces x es falsa.
- Una asignación de verdad es satisfaciente para la fórmula Φ si hace que la fórmula sea verdadera.
- Una fórmula Φ se dice que es satisfactible si existe una asignación de verdad satisfaciente para Φ.

[2]Problema de decisión Np-completo [CLR90].

A continuación se define el problema de SAT.

Satisfabilidad (SAT)

Entrada: Un conjunto U de variables booleanas y una colección C de cláusulas sobre U.

Pregunta: ¿Existe una asignación de verdad sobre U que satisfaga todas las cláusulas en C?.

El algoritmo probabilístico para este problema es:

Inicio{entrada: C}
  $T \leftarrow$ asignarle valores de verdad
 para $i \leftarrow 1$ **hasta** r **hacer**
  si no hay clausula insatisfactible **entonces**
   devolver "la colección C es satisfaciente"
  sino
   Tomar cualquier clausula no satisfaciente, escoger cualquier literal
   de la clausula seleccionada aleatoriamente; cambiarle el valor de
   de verdad, actualizar T
 devolver "la colección C es probablemente insatisfaciente"
Fin.

El parámetro r es el número de ejecuciones que realizará el algoritmo; lo que se busca es que r sea un polinomio en el número de variables. Note que no se especifica cómo se escoge una cláusula no satisfactible por los valores de verdad actuales en T, o cómo se escoge la asignación de verdad inicial; solamente se necesita aleatoridad para escoger el literal al cual se le cambiará el valor de verdad. Se elige un literal aleatoriamente entre aquellos de la cláusula seleccionada, cada uno con igual probabilidad de ser elegido. El valor de verdad del literal seleccionado está en T, pero como este es cambiado, se dice que se ha actualizado T; este proceso se repite hasta que una asignación de verdad satisfaciente sea encontrada o hasta que los r cambios (actualizaciones) sean ejecutados. El algoritmo descrito anteriormente es llamado *Random walks (caminos aleatorios)*.

Si la colección no es satisfactible el algoritmo responde "correctamente", porque no es posible que en alguno de los r pasos el algoritmo encuentre una asignación de verdad satisfaciente para la colección. Pero, ¿qué si la colección es satisfactible?. En este caso, si se permiten muchas (exponenciales) repeticiones, con una probabilidad muy alta, se encontrará una asignación de verdad satisfaciente para C.

La pregunta realmente importante es: ¿cuál es la probabilidad de que una asignación de verdad satisfaciente sea encontrada cuando r es polinomial en el número de variables booleanas?.

Cuando el algoritmo Random walks (caminos aleatorios) es aplicado a instancias satisfactibles de 3SAT[3], éste se ejecuta débilmente en todos los posibles casos [Pap94]. Pero como se verá en el siguiente teorema, cuando el algoritmo es aplicado a instancias satisfactibles de 2SAT[4] el algoritmo se ejecuta de manera eficiente.

El problema de 2Satisfabilidad(2SAT) se define de la siguiente manera:

2satisfabilidad (2SAT)

Entrada: Un conjunto U de variables booleanas y una colección $C = \{c_1, c_2, \ldots, c_m\}$ de cláusulas sobre U tal que $|c_i| = 2$ para toda $i = 1, 2 \ldots, m$

Pregunta: ¿Existe una asignación de verdad sobre U que satisfaga todas las cláusulas en C?.

Teorema 1.3.1. *Si el algoritmo Random Walks con $r = 2n^2$ es aplicado a instancias satisfacientes de 2SAT con n variables, entonces la probabilidad de que una asignación de verdad satisfaciente sea encontrada es por lo menos $\frac{1}{2}$.*

Demostración. Sean $\widehat{T}$ la asignación de verdad satisfaciente que devuelve el algoritmo y $t(i)$ una función de i que denota el número esperado de repeticiones del algoritmo hasta

[3]Se define similar a 2SAT, donde la cardinalidad de cada c_i es igual a 3.

[4]2SAT, es un problema que pertenece a la clase P [Pap94, PC94].

que una asignación de verdad satisfaciente sea encontrada. Note que, T y $\widehat{T}$ son distintas en exactamente i valores.

¿Que hacer con lo que se sabe acerca de $t(i)$?. Primero se tiene que $t(0) = 0$, porque si el algoritmo comienza con alguna asignación de verdad satisfaciente, en particular $\widehat{T}$, no es necesario hacer ningún cambio(actualizar). De lo contrario es necesario actualizar al menos una vez, cuando el paso de actualización es realizado, se escoge entre 2 literales de una cláusula que no es satisfaciente por la actual T y alguno de estos dos literales es ahora verdadero sobre $\widehat{T}$, porque $\widehat{T}$ satisface todas las cláusulas. Por lo tanto, para actualizar una variable aleatoriamente escogida de una cláusula no satisfaciente, se tiene al menos $\frac{1}{2}$ de posibilidades de cambio para encontrar $\widehat{T}$.

Así, para $0 < i < n$ se tiene la desigualdad:

$$t(i) \leq \frac{1}{2}(t(i+1) + t(i-1)) + 1 \tag{1.1}$$

Donde se estan sumando los cambios justamente hechos. La desigualdad 1.1 es válida ya que se puede dar cualquiera de los siguientes casos:

1. La asignación de verdad inicial T es satisfaciente para C, o en particular es $\widehat{T}$, entonces (1.1) se cumple ya que como no es necesario actualizar se tiene que $t(0) < 1$.

2. T difiera de $\widehat{T}$ en ambos literales en algunas cláusulas.

También se tiene que $t(n) \leq t(n-1) + 1$ para $i = n$, porque T puede tener a lo más n valores de verdad distintos con $\widehat{T}$, ya que se tienen solamente n variables.

Ahora será considerado otro caso, el cual dará la posibilidad de encontrar alguna asignación de verdad satisfaciente o una cláusula donde T y $\widehat{T}$ difieran en ambos literales. Es decir, considérese la función $x(i)$ tal que:

$$x(0) = 0$$

$$x(i) = \frac{1}{2}(x(i+1) + x(i-1)) + 1 \tag{1.2}$$

y

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

Por la forma como ha sido definida $x(i)$, es claro, que $t(i) \leq x(\text{i})$ para toda $i = 1, 2 \ldots, n$. Las $x(i)$ serán calculadas, sumando todas la ecuaciones a la vez. Como se mostrará a continuación, por claridad en los cálculos las $x(i)$ serán notadas por x_i. Entonces la ecuación (1.2) sería:

$$x_0 = 0 \qquad x_i = x_{i+1} + x_{i-1} + 2 \qquad x_n = x_{n-1} + 1$$

i	Ecuación		
1	$2x_1$	$=$	$x_2 + x_0 + 2$
2	$2x_2$	$=$	$x_3 + x_1 + 2$
3	$2x_3$	$=$	$x_4 + x_2 + 2$
$\vdots$	$\vdots$		$\vdots$
n-1	$2x_{n-1}$	$=$	$x_n + x_{n-2} + 2$
n	x_n	$=$	$x_{n-1} + 1$

sumando lado a lado las ecuaciones se tiene que.

$$2\sum_{i=1}^{n-1} x_i = \sum_{i=1}^{n-1} x_i + \sum_{i=1}^{n-1} x_i - x_1 + 2(\text{n-1}) + 1$$

De ahí, $x_1 = 2n - 1$

De igual manera sumando desde $n = 2$ se tiene que $x_2 = 4n-4$; desde $n = 3$, $x_3 = 6n-9$, de donde se cumple que $x_i = 2ni - i^2$. Como se esperaba en el peor de los casos $x_n = n^2$

Como x(i) es una función creciente de i, se tiene que el número necesario de repeticiones para encontrar una asignación de verdad satisfaciente es $t(i) \leq x(i) \leq x(n) = n^2$.

Por lo tanto, t(i) $\leq n^2$, esto es, el número de repeticiones para encontrar una asignación de verdad satisfaciente es a lo más n^2.

Para completar la demostración del teorema se tendrá en cuenta el siguiente lema:

Lema 1. *X es una variable aleatoria que toma valores positivos, con esperanza finita, entonces para cualquier $k > 0$ se tiene que $prob[x > k\varepsilon(x)] \leq \frac{1}{k}$.*

Demostración. p_i denota la probabilidad de que $x = i$, entonces,

$$\varepsilon(\mathrm{x}) = \sum ip_i = \sum_{i \geq k\varepsilon(\mathrm{x})} ip_i + \sum_{i < k\varepsilon(\mathrm{x})} ip_i$$

Pero,

$$\sum_{k\varepsilon(x) < i} ip_i \geq \sum_{i > k\varepsilon(x)} k\varepsilon(x)p_i = k\varepsilon(\mathrm{x}) \sum_{i > k\varepsilon(x)} p_i$$

Además, se sabe que.

$$k\varepsilon(\mathrm{x}) \sum_{i < k\varepsilon(x)} p_i = k\varepsilon(\mathrm{x})prob[x > k\varepsilon(\mathrm{x})]$$

De ahí que

$$\varepsilon(\mathrm{x}) > k\varepsilon(\mathrm{x})prob[x > k\varepsilon(\mathrm{x})]$$

De donde, $prob[x > k\varepsilon(\mathrm{x})] < \frac{1}{k}$ $\diamondsuit$

Por lo tanto, del lema anterior con $k = 2$ se tiene que la probabilidad de no encontrar una asignación de verdad satisfaciente, es decir se ejecuten más del número esperado de pasos, es menor que $\frac{1}{2}$. De ahí que la probabilidad de que una asignación de verdad satisfaciente sea encontrada al aplicar el algoritmo Random Walks a instancias satisfactibles de 2SAT, es por lo menos $\frac{1}{2}$, con $r = 2n^2$. $\diamondsuit$

1.3.2. Identidades polinomiales

Otro uso importante de la aleatoriedad es en la verificación de identidades polinomiales (en la siguiente sección se mostrará como esta técnica puede ser aplicada en teoría de grafos). En particular se considerará el problema del cero polinomial, el cual se define a continuación.

Problema del cero polinomial
Entrada: Un polinomio multivariado $p\big(x_1, x_2, \ldots, x_n\big)$ de grado d[6].

Pregunta: ¿p es identicamente cero?.

Una idea interesante para resolver este problema es, dado un polinomio multivariado (Suponga que es un polinomio en n variables) seleccionar aleatoriamente una n -tupla $(a_1, a_2, \ldots, a_n)$ de enteros y evaluarla en el polinomio; si el resultado obtenido es distinto de cero, se puede asegurar que el polinomio no es idénticamente cero. Pero, si el resultado obtenido es cero el polinomio dado podría ser siempre cero, pero, también se puede tener la mala suerte de que la n-tupla $(a_1, a_2, \ldots, a_n)$ seleccionada sea una raíz del polinomio. El siguiente teorema dice el número de n-tuplas de enteros que son raíces de un polinomio que es no idénticamente cero, en el intervalo $-kd$ y kd donde $k \in \mathbb{Z}^+$ y d es el grado del polinomio.

Teorema 1.3.2. *Sea $p(x_1, x_2, \ldots, x_n)$ un polinomio multivariado de grado d. Si p no es idénticamente cero, entonces el número de n-tuplas $(a_1, a_2, \ldots, a_n)$ de enteros entre $-kd$ y kd, $k \in \mathbf{Z}^+$ tal que $p(a_1, a_2, \ldots, a_n) = 0$, es a lo más $nd(2kd+1)^{n-1}$*

Demostración. La demostracción se hará por inducción sobre n:

- Para $n = 1$:

  Sea p un polinomio en una sola variable de grado d, esta variable será llamada x. Luego p es de la forma

  $$p(x) = c_0 + c_1 x + \cdots + c_d x^d$$

  Entonces, p tiene a lo más d ceros, esto es, el teorema se cumple para $n = 1$.

[6]El grado de un polinomio multivariado es el más grande de los grados de sus variables

- Suponga que el teorema es válido para $n-1$ y se probará para n.

  Sea $p(x_1, x_2, \ldots, x_n)$ un polinomio multivariado de grado d . Note que, $p(x_1, x_2, \ldots, x_n)$ puede ser escrito como un polinomio en una sola variable, donde los coeficientes son polinomios en las restantes $n-1$ variables; p será escrito como un polinomio en x_1, esto es.

  $$p(x_1) = b_0 + b_1 x_1 + \ldots + b_d x_1^d$$

  Donde b_i son polinomios en las variables $(x_2, x_3, ..., x_n)$

  Sea $p' = b_i'$, donde b_i' es el coeficiente de grado más alto de $p(x_1)$.

  Para cualquier $(n-1)$-tupla $(a_2, a_3, \ldots, a_n)$ de coeficientes enteros entre $-kd$ y kd, se tendrá que $p'(a_2, a_3, \ldots, a_n) = 0$ ó $p'(a_2, a_3, \ldots, a_n) \neq 0$.

  1. Si $p'(a_2, a_3, \ldots, a_n) = 0$ se puede dar que $p(x_1, a_2, \ldots, a_n) = 0$ para todos los posibles valores de x_1 entre $-kd$ y kd los cuales son $(2kd+1)$

     Ahora, por la hipótesis de inducción, el número de $(n-1)$-tuplas $(a_2, \ldots, a_n)$ de enteros entre $-kd$ y kd tal que $p'(a_2, a_3, \ldots, a_n) = 0$ es a lo más, $(n-1)d$ $(2kd+1)^{n-2}$. De ahí, $p(x_1, a_2, \ldots, a_n)$ será cero para a lo más.

     $$(2kd+1)(n-1)d(2kd+1)^{n-2} = (n-1)d(2kd+1)^{n-1} \tag{1.3}$$

     $n-tuplas$ de enteros entre $-kd$ y kd

  2. Si $p'(a_2, a_3, \ldots, a_n) \neq 0$ se tiene que $p(x_1, a_2, \ldots, a_n)$ puede ser cero, a lo más para d valores, ya que, p es polinomio de grado a lo más d.

     Como el número de $(n-1)$-tuplas de enteros entre $-kd$ y kd es igual a $(2kd+1)^{n-1}$, entonces p es cero para a lo más.

     $$d(2kd+1)^{n-1} \tag{1.4}$$

     n-tuplas de enteros entre $-kd$ y kd

Por lo anterior, el número de ceros de p es a lo más

$$(2kd+1)(n-1)d(2kd+1)^{n-2} + d(2kd+1)^{n-1}$$

$$
\begin{aligned}
&= (n-1)d(2kd+1)^{n-1} + d(2kd+1)^{n-1} \\
&= nd(2kd+1)^{n-1} - d(2kd+1)^{n-1} + d(2kd+1)^{n-1} \\
&= nd(2kd+1)^{n-1}
\end{aligned}
$$

Por lo tanto, el teorema es válido para toda $n \in \mathbf{Z}^+$ $\diamondsuit$

Como un caso particular del teorema anterior se tiene el siguiente corolario.

Corolario 1.1. *Sea $p(x_1, x_2, \ldots, x_n)$ un polinomio multivariado de grado d. Si p no es idénticamente cero, entonces, el número de n-tuplas $(a_1, a_2, \ldots, a_n)$ de enteros entre $-nd$ y nd tal que $p(a_1, a_2, \ldots, a_n) = 0$, es a lo más $nd(2nd+1)^{n-1}$*

Demostración. Del teorema anterior tomando $k = n$. $\diamondsuit$

Puesto que el número de n-tuplas de enteros entre $-nd$ y nd es igual a $(2nd+1)^n$, seleccionando una $n-tupla$ de éstas aleatoriamente y aplicando el corolario anterior se tiene que.

$$
\begin{aligned}
Prob[p(a_1, a_2, \ldots, a_n) = 0] &= \frac{nd(2nd+1)^{n-1}}{(2nd+1)^n} \\
= \frac{nd}{2nd+1} = \frac{nd}{nd(2+\frac{1}{nd})} &= \frac{1}{(2+\frac{1}{nd})} < \frac{1}{2}
\end{aligned}
$$

Esto es, la probabilidad de que al evaluar una n-tupla seleccionada aleatoriamente en un polinomio que no es idénticamente cero, sea una raíz del polinomio, es a lo más, $\frac{1}{2}$.

De lo anterior, se obtiene el siguiente algoritmo probabilístico para el problema del cero polinomial.

Inicio{Entrada:$p(x_1, x_2, \ldots, x_n)$}
  $d \leftarrow$ *grado* de p
 para $i \leftarrow 1$ **hasta** n **hacer**

$a_i \leftarrow Random(-nd, nd)$

si $p(a_1, a_2, \ldots, a_n) \neq 0$ **entonces**

responder "El polinomio no es idénticamente cero"

sino

responder "Probablemente el polinomio es idénticamente cero"

Fin.

Note que el orden de este algoritmo depende del tiempo necesario para calcular el grado de p y la longitud de n; para calcular el grado de p, utilizando un "buen " algoritmo ordenación se tiene un tiempo de $n \lg n$. Luego el orden del algoritmo anterior es $n \lg n + n = n(1 + \lg n)$ lo cual es un polinomio en n.

Así, el algoritmo anterior es un algoritmo probabilístico tiempo polinomial con error de probabilidad menor que $\frac{1}{2}$, es decir, es un buen algoritmo para el problema del cero polinomial.

1.3.3. Verificando Matching en grafos

El tercer ejemplo de algoritmos probabilísticos que se verá es un problema en teoría de grafos, el problema del matching perfecto[5].

problema del matching perfecto

Entrada: Un grafo $G = (N, E)$

Pregunta: ¿G tiene un matching perfecto?, esto es, ¿existe un subconjunto $E' \subseteq E$ tal que, para cualquier nodo u de G existe una y sólo un árista que tiene a u como uno de sus puntos finales.?

Ejemplo 1.

En el grafo G formado por el conjunto de vértices {1, 2, 3, 4, 5, 6} que se muestra en la figura 1.1 hay 3 matching perfectos, los conjuntos E'_1, E'_2, E'_3 donde:

[5]El problema del matching perfecto pertenece a la clase P, es decir, existen algoritmos deterministicos tiempo polinomial que lo resuelven, pero no son tan simples como el algoritmo probabilístico que se verá.

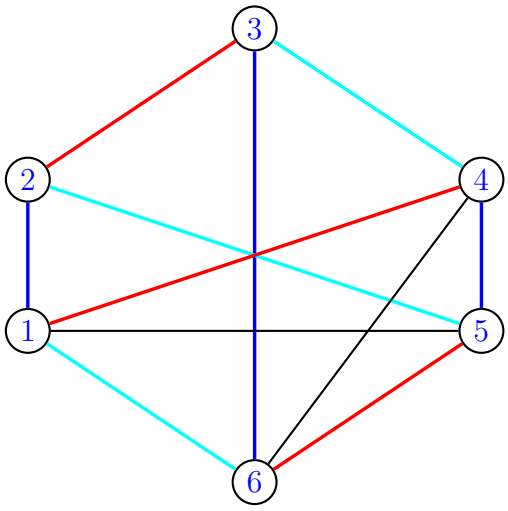


Figura 1.1: Grafo G con conjunto de vértices $\{1, 2, 3, 4, 5, 6\}$

$$
\begin{aligned}
E'_1 &= \{\{1,2\},\{3,6\},\{5,4\}\} \\
E'_2 &= \{\{1,6\},\{2,5\},\{3,4\}\} \\
E'_3 &= \{\{1,4\},\{3,2\},\{5,6\}\}
\end{aligned}
$$

La idea para resolver el problema es llevarlo al cero polinomial, sección anterior(1.3.2). El siguiente teorema da una condición "algebraica"necesaria y suficiente para que un grafo G tenga un matching perfecto.

Teorema 1.3.3. *Sean G un grafo con conjunto de vertices* $\{1, 2, \ldots, n\}$ *y* $A \in M_{n\times n}$ *definida por*

$$
a_{ij} = \begin{cases} x_{ij} & \text{si } i \text{ es adyacente a } j \text{ y } j > i, \\ -x_{ij} & \text{si } i \text{ es adyacente a } j \text{ y } j < i, \\ 0 & \text{en otro caso} \end{cases}
$$

Entonces, G tiene un matching perfecto si, y sólo si, el determinante de A no es idénticamente cero.

Demostración. Recuerde que el determinante de una matriz A está dado por:

$$detA = \sum_{\pi} \sigma_{\pi} \prod_{i=1}^{n} a_{i\pi(i)}$$

Donde π denota las permutaciones del conjunto $\{1, 2, \ldots, n\}$ y σ_π es 1(-1) si π es el producto de un número par(impar) de transposiciones[6]. Note que para cualquier permutación π, $\prod_{i=1}^{n} a_{i\pi(i)} \neq 0$ si, y sólo si, i es adyacente a $\pi(i)$ para $1 \leqslant i \leqslant n$, esto es cualquier permutación π tal que $\prod_{i=1}^{n} a_{i\pi(i)} \neq 0$ corresponde a un subgrafo G_π de G el cual consta de los lados $\langle i, \pi(i) \rangle$ para $1 \leqslant i \leqslant n$.

Observe que todas las permutaciones π tales que G_π contiene al menos un ciclo impar no contribuyen al determinante de A, puesto que, tales permutaciones pueden ser agrupadas en pares que se cancelan con otras. De la siguiente manera, se asocia la permutación π con otra permutación π' con un ciclo impar contrario, puesto que $\prod_{i=1}^{n} a_{i\pi(i)} = -\prod_{i=1}^{n} a_{i\pi'(i)}$ y $\sigma_\pi = \sigma'_\pi$, la contribución total al determinante de π y π' es cero.

En consecuencia, sólo se deben considerar permutaciones cuyos subgrafos correspondan sólo a ciclos pares. Con cada una de tales permutaciones se puede asociar otra permutación π^r en la cual todos los ciclos son contrarios, obsérvese que $\sigma_\pi = \sigma_\pi^r$ porque los ciclos son de igual longitud, pero alguno en orden distinto.

Se notará por E' un matching perfecto, y se define $t_{E'}$ como el producto de las x correspondientes a los lados de E'. Entonces se dan los siguientes dos casos.

1. Si $\pi = \pi^r$. En este caso, G_π consiste solamente de ciclos de longitud 2 y π corresponde al matching perfecto E' tal que $\prod_{i=1}^{n} a_{i\pi(i)} = (t_{E'})^2$

2. Si $\pi \neq \pi^r$. En este caso ambos π y π^r corresponden a la unión de dos matching perfectos E' y E'', obtenidos selecionando alternadamente lados con los ciclos tales que $\prod_{i=1}^{n} a_{i\pi(i)} + \prod_{i=1}^{n} a_{i\pi^r(i)} = 2t_{E'}t_{E''}$

Para todo matching perfecto E' que existe en el grafo G, se pueden encontrar π y π^r tales que $\prod_{i=1}^{n} a_{i\pi(i)} = (t_{E'})^2$ y $\prod_{i=1}^{n} a_{i\pi(i)} + \prod_{i=1}^{n} a_{i\pi^r(i)} = 2t_{E'}t_{E''}$. De ahí, el determinante de

[6]Una transposición es un permutación de dos elementos, o de otra manera, es un ciclo de longitud 2.

A es igual a

$$detA = t^2_{E'_1} + t^2_{E'_2} + \ldots + t^2_{E'_h}$$

$$\begin{aligned} + \quad & 2t_{E'_1}t_{E'_2} + 2t_{E'_1}t_{E'_3} + \cdots + 2t_{E'_1}t_{E'_h} + 2t_{E'_2}t_{E'_3} + 2t_{E'_2}t_{E'_4} + \cdots + 2t_{E'_{h-1}}t_{E'_h} \\ = \quad & (t_{E'_1} + t_{E'_2} + \ldots + t_{E'_h})^2 \end{aligned}$$

donde E'_i denota el i-ésimo matching perfecto, y así el determinante de A es idénticamente cero si, y sólo si, G no tiene matching perfecto. $\diamondsuit$

En conclusión, el problema de decidir si un grafo G tiene un matching perfecto ha sido reducido al problema de verificar si un polinomio es idénticamente cero, el cual se sabe de la sección anterior 1.3.2, que se resuelve probabilísticamente en tiempo polinomial. Para ilustrar el teorema 1.3.3 se da el siguiente ejemplo.

Ejemplo 2.

$$A = \begin{pmatrix} 0 & x_{12} & 0 & x_{14} \\ -x_{12} & 0 & x_{23} & x_{24} \\ 0 & -x_{23} & 0 & x_{34} \\ -x_{14} & -x_{24} & -x_{34} & 0 \end{pmatrix}$$

Aplicando el teorema anterior a él grafo G mostrado en la figura 1.2, cuya matriz A se muestra arriba. se tiene que el determinante de A es igual a:

$$detA \; = \; (x_{12}x_{34} \; + \; x_{14}x_{23})^2$$

En efecto, las permutaciones no nulas de A que corresponden a subgrafos de ciclos pares son: π_1, π_2, π_3, y, π_3^r. Las cuales se muestran en la tabla 1.3.3. Entonces,

$$\begin{aligned} detA \quad &= \quad \sum_{\pi} \sigma_\pi \prod_{i=1}^{n} a_{i\pi(i)} \\ &= \quad \prod_{i=1}^{4} a_{i\pi_1(i)} \; + \; \prod_{i=1}^{4} a_{i\pi_2(i)} \; - \; (\prod_{i=1}^{4} a_{i\pi_3(i)} \; + \; \prod_{i=1}^{4} a_{\pi_3^r(i)}) \end{aligned}$$

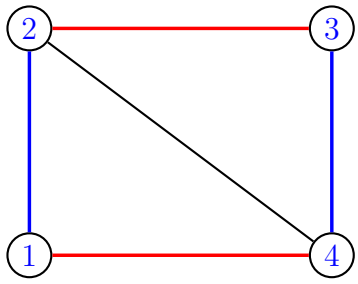


Figura 1.2: Grafo G con conjunto de vértices $\{1, 2, 3, 4\}$

$$= (x_{12}x_{34})^2 + (x_{14}x_{23})^2 - (-2x_{12}x_{34}x_1x_{23})$$
$$= (x_{12}x_{34})^2 + 2x_{12}x_{34}x_{14}x_{23} + (x_{14}x_{23})^2 = (x_{12}x_{34} + x_{14}x_{23})^2.$$

Permutaciones de ciclos pares	Ciclo	Ciclo contrario	Permutacioń obtenida
$(2,1,4,3) = \pi_1$	$(1,2)(3,4)$	$(2,1)(4,3)$	$(2,1,4,3)$
$(4,3,2,1) = \pi_2$	$(1,4)(2,3)$	$(4,1)(3,2)$	$(4,3,2,1)$
$(2,4,1,3)$	$(1,2,4,3)$	$(3,4,2,1)$	$(3,1,4,2)$
$(3,1,4,2)$	$(1,3,4,2)$	$(4,3,1,2)$	$(2,4,1,3)$
$(3,4,1,2)$	$(2,4)(1,3)$	$(4,2)(3,1)$	$(3,4,1,2)$
$(3,4,2,1)$	$(1,3,2,4)$	$(2,3,1,4)$	$(4,3,1,2)$
$(2,3,4,1) = \pi_3$	$(1,2,3,4)$	$(2,1,4,3)$	$(4,1,2,3)$
$(4,1,2,3) = \pi_3^r$	$(1,4,3,2)$	$(2,3,4,1)$	$(2,3,4,1)$

tabla 1.1: permutaciones que corresponden a ciclos pares del grafo G de la figura 1.2

2. CLASES DE ALGORITMOS PROBABILÍSTICOS

Avanzando en el estudio de los algoritmos probabilísticos, surgen algunas preguntas como las siguientes: ¿Puede la teoría de la probabilidad ayudar a resolver problemas puramente deterministas?, en caso afirmativo, ¿de qué forma se aplica la aleatoriedad a los algoritmos deterministas?, y ¿cuáles son las ventajas de usar estos algoritmos?.

La respuesta a las preguntas anteriores es que no solamente la teoría de la probabilidad sino más concretamente la teoría de los algoritmos probabilísticos ayuda a resolver problemas puramente deterministas sino que, en muchas ocasiones, es la *única manera de hacerlo*. Él cómo hacerlo y de que forma suele ser, generalmente, mediante la aplicación de los algoritmos probabilísticos conocidos: **Algoritmos de Monte Carlo** y **algoritmos de las Vegas**[7].

La ventaja que presentan estos algoritmos es su costo computacional relativamente bajo y la facilidad en la implementación. Además, en muchos casos, al introducir elecciones aleatorias en un algoritmo se puede obtener un mejor rendimiento que al aplicar el algoritmo determinístico puro.

En esta sección se estudiarán dos tipos de algoritmos probabilísticos llamados **Algoritmos de Monte Carlo** y **algoritmos de las Vegas**. Los algoritmos de Monte Carlo aseguran un tiempo fijo de ejecución pero no garantizan que la respuesta sea correcta, aunque lo

[7]En algunos de los textos consultados se muestra una tercera clase de algoritmos probabilísticos, los algoritmos probabilísticos numéricos. En este documento no se estudiará esta clase porque dicha clase está contenida en la clase de los algoritmos de Monte Carlo.

puede ser con alta probabilidad. No se puede saber si la respuesta proporcionada por un algoritmo de Monte Carlo es correcta o no, ya que el algoritmo nunca indica cuándo se equivoca. Al contrario, los algoritmos de las Vegas son más eficientes y confiables, porque en lugar de dar una respuesta incorrecta, ellos devuelven un "no sé", siempre que no estén seguros. Lo que no garantizan es el tiempo total de ejecución, aunque con alta probabilidad, éste será bajo. Para mayor claridad serán presentados 2 ejemplos de cada una de las clases de algoritmos probabilísticos mencionadas anteriormente.

2.1. ALGORITMOS DE MONTE CARLO

Existen problemas para los cuales no se conoce ningún algoritmo eficiente (Probabístico o determinístico) que sea capaz de obtener una respuesta correcta en todas las ocasiones. Los algoritmos de Monte Carlo cometen ocasionalmente un error, pero encuentran la solución correcta con una alta probabilidad sea cual sea el caso considerado. Sin embargo, no suelen dar ningún aviso cuando el algoritmo comete un error.

Informalmente, un algoritmo de Monte Carlo es un algoritmo probabilístico[8] que retorna respuestas exactas, con una alta probabilidad de ser la correcta.

Sea ρ un número real tal que $0 < \rho < 1$, se dice que un algoritmo de Monte Carlo es ρ-correcto si devuelve una respuesta correcta con probabilidad no menor que ρ, sea cual sea el caso considerado. En algunos casos se permite que ρ dependa del tamaño de la entrada, pero nunca de la entrada en sí; la característica más importante de los algoritmos de Monte Carlo es que suele ser posible reducir arbitrariamente la probabilidad de error a costa de un ligero aumento del tiempo de ejecución. Se considerarán 2 ejemplos, el primero es la **verificación de la multiplicación de matrices** el cual tiene escaso interés práctico, pero tiene la ventaja de su sencillez y será utilizado para presentar las nociones claves; el segundo es el **verificador de composición**, el cual tiene un indudable interés práctico.

2.1.1. Verificación de la multiplicación de matrices

Se Considerará el problema de verificar la multiplicación de matrices, el cual esta dado por:

[8]La definición formal de algoritmo probabilístico se dará en el capítulo 3

Entrada: tres matrices $n \times n$, A, B y C

Pregunta: ¿Es $C = AB$?

Este problema pertenece a la clase P, es decir, existen algoritmos determinísticos tiempo polinomial que lo resuelven y La forma como ellos proceden es la siguiente multiplicando A por B y comparando el resultado obtenido con C; empleando el algoritmo de multiplicación matricial asintóticamente mas rápido se requiere un tiempo que está en $O(n^{2,5})$.[PC94]

¿Será Posible dar un algoritmo que emplee menor tiempo, si se admite una pequeña probabilidad de error?. La respuesta es que con un tiempo de complejidad que está en $O(n^2)$, se podrá resolver este problema con una probabilidad de error de a lo más un $\frac{1}{2}$. A continuación se mostrará como construir este algoritmo.

Suponga que $C \neq AB$. Sea $D = AB - C$, como $C \neq AB$ entonces $D \neq 0$, es decir, D no es idénticamente cero. Sea i un entero tal que la i-ésima columna de D contiene, al menos un elemento no nulo. Considere cualquier subconjunto $S \subseteq \{1, 2, \ldots, n\}$. El término $\sum_s(D)$, representará el vector de longitud n que se obtiene sumando componente a componente las columnas de D indexadas por los elementos de S y colocando ceros en las restantes posiciones .

El conjunto S' es definido de la siguiente manera: $i \in S'$ si, y sólo si, $i \notin S$. Note que como la i-ésima columna de D no es idénticamente cero $\Sigma_S(D)$ y $\Sigma_{S'}(D)$ no pueden ser nulas simultáneamente, como se ve en el ejemplo siguiente.

Ejemplo 3.

Consideremos las matrices:

$$A = \begin{pmatrix} 8 & 5 & -1 & 0 \\ 1 & 0 & 3 & 7 \\ -10 & 8 & 2 & 1 \\ 6 & 0 & 1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 2 & 3 & 6 \\ -1 & 7 & 5 & 3 \\ 3 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 0 & 50 & 45 & 63 \\ 10 & 5 & -1 & -1 \\ -12 & 38 & 10 & 37 \\ 9 & 13 & 19 & 36 \end{pmatrix}$$

$$AB = \begin{pmatrix} 0 & 50 & 48 & 63 \\ 10 & 5 & -1 & -1 \\ -12 & 38 & 14 & 37 \\ 9 & 13 & 19 & 36 \end{pmatrix} \quad \text{y} \quad D = \begin{pmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 6 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Tomando los conjuntos $S = \{1, 3\}$ y $S' = \{2, 4\}$, se tiene que $\Sigma_S(D) = (6, 0, 7, 0)$ y $\Sigma_{S'}(D) = (0, 0, 0, 0)$

Suponga que S es seleccionado aleatoriamente entre 1 y n. Para todo j entre 1 y n se puede dar que: $j \in S$ ó $j \notin S$. Dado que es tan probable que $i \in S$ como que $i \notin S$, la probabilidad de que $\Sigma_S(D) \neq 0$ es de al menos un $\frac{1}{2}$. Esto es, la probabilidad de que $i \in S$ es de al menos $\frac{1}{2}$

Por otra parte, $\sum_S(D)$ siempre es cero si $C = AB$, ya que D es la matriz idénticamente cero($D = 0$). Lo anterior da una idea para comprobar si $C = AB$ o no, calculando $\sum_S(D)$ para un subconjunto S seleccionado aleatoriamente y verificar si la igualdad $\Sigma_S(D) = 0$ se cumple. Pero, ¿Cómo se puede hacer esto sin calcular D (y por tanto AB)?

Para solucionar el problema anterior se considerará el vector binario x de longitud n tal que $x_i = 1$ si $i \in S$ y 0 en otro caso. $\Sigma_S(D) = xD$, en efecto, $\Sigma_S(D)$ es el vector de longitud n que representa la suma componente a componente de las columnas en D indexadas por los elementos en S, es decir, $\Sigma_S(D) = \Sigma_{i \in S} \Sigma_{j=1}^{n} d_{ij}$. Además $xD = \Sigma_{i=1}^{n} \Sigma_{j=1}^{n} x_i d_{ij}$. Pero $x_i = 1$ si $i \in S$ entonces $xD = \Sigma_{i \in S} \Sigma_{j=1}^{n} dij = \Sigma_S(D)$.

Ya se había dicho que para comprobar si $C = AB$, es suficiente calcular $\Sigma_S(D)$ para un vector S seleccionado aleatoriamente y verificar si $\Sigma_S(D) = 0$, pero se probó que $\Sigma_S(D) = xD = x(AB - C)$, luego la comprobación corresponde a verificar si $x(AB - C)$ es idénticamente cero o no, o de manera equivalente, verificar si $xAB = xC$ para algún vector binario x seleccionado aleatoriamente; podría pensarse que el tiempo necesario para calcular xAB es más que el tiempo necesario para calcular AB, pero esto no es así.

Recuerde que el tiempo necesario para calcular una multiplicación matricial encadenada depende del orden en que se multipliquen las matrices [BB97], en éste caso sólo se necesita un tiempo que está en $O(n^2)$ para calcular xAB en la forma $(xA)B$. Lo anterior sugiere un algoritmo probabilístico para verificar si $C = AB$. Veamos el algoritmo.

Función *Verificarmultiplicación*(Entrada: A, B, C, n)
Inicio
Para $i \leftarrow$1 hasta n **hacer**
$x_i \leftarrow random(0, 1)$
si $(xA)B = xC$ **entonces**
Responder "Verdadero, (probablemente $C = AB$)"
sino
Responder "falso, (C no es el producto de AB)"
fin.

La probabilidad de error de este algoritmo es de a lo más $\frac{1}{2}$ ya que si la matriz D contiene sólo una columna con un termino no nulo, suponga que la columna que contiene el termino no nulo es la i-ésima se puede dar que $x_i = 1$ ó $x_i = 0$ cada uno con probabilidad $\frac{1}{2}$ de ser seleccionado. Si $x_i = 0$ entonces $xD = 0$, lo cual equivale a $xAB = xC$. En este caso el algoritmo devuelve "Verdadero, (probablemente $C = AB$)". Pero, la respuesta correcta para este caso es falso, ya que D tiene una columna con un elemento no nulo. Así la probabilidad de error del algoritmo es de a lo más $\frac{1}{2}$; cuando el algoritmo devuelve "falso", es decir, $C \neq AB$ se puede estar seguros que la respuesta es correcta, esto es, la probabilidad de error en este caso es cero.

Ejemplo 4.
Consideremos las matrices $A, B, C \in M_{3\times3}$

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 8 & 2 \\ 3 & 2 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 4 & 6 & 7 \\ 6 & 1 & 6 \\ 7 & 6 & 4 \end{pmatrix} \quad C = \begin{pmatrix} 37 & 26 & 31 \\ 30 & 32 & 70 \\ 31 & 26 & 35 \end{pmatrix}$$

Una llamada al algoritmo *Verificarmultiplicación*(A, B, C, n) podría tomar el vector $x = (1, 0, 1)$ en cuyo caso $xA = (4, 4, 4)$, y multiplicando éste resultado por B se tiene que $(xA)B = (68, 52, 66)$. Este resultado se compara con $xC = (68, 52, 66)$, con el vector $x = (1, 0, 1)$ seleccionado aleatoriamente el algoritmo devuelve le valor "*Verdadero*", puesto que $xAB = xC$.

Otra llamada a *Verificarmultiplicación*(A, B, C, n) podría seleccionar el vector $x = (1, 1, 0)$ en éste caso los cálculos son: $xA = (3, 10, 5)$, $(xA)B = (107, 58, 101)$, $xC = (37, 58, 101)$ y el algoritmo devuelve "*falso*"y en este caso tendríamos más suerte ya que el hecho de que exista x tal que $xAB \neq xC$ es una demostración suficiente de que $AB \neq C$.

El orden del algoritmo anterior depende del tiempo empleado para realizar la multiplicación encadenada de matrices, el cual es, $O(n^2)$. Es decir, el algoritmo anterior es un algoritmo polinomial de Monte Carlo con probabilidad de error de a lo más $\frac{1}{2}$, cuando el algoritmo devuelve $C = AB$. Recuerde que la característica principal de este tipo de algoritmos es que la probabilidad de error se puede hacer arbitrariamente pequeña con un ligero aumento en el tiempo de ejecución.

Después del análisis anterior se pensaría en ejecutar varias veces el algoritmo sobre la misma entrada con escogencias aleatorias independientes del vector x. Si se obtiene respuesta "*falso*"incluso en una sola ocasión, se puede concluir que $AB \neq C$ independientemente de el número de veces que se obtenga la respuesta "*verdadero*". Considere el siguiente algoritmo en el cual k es el nuevo parámetro, k indica el número de veces que será repetido el algoritmo *VerificarMultiplicación*.

Función*RepetirVerificarmultiplicación*(A, B, C, n, k)
Inicio
cont $\leftarrow 1$
tempo $\leftarrow$ *Verificarmultiplicación*(A, B, C, n)
mientras $(tempo = verdadero)$ **y** $(cont < k)$ **hacer**
inicio

$cont \leftarrow cont + 1$

$tempo \leftarrow$ *Verificarmultiplicación*(A, B, C, n)

fin

si $tempo = verdadero$ **entonces**

Responder "Verdadero"

sino

Responder "falso"

Fin.

Para analizar la probabilidad de error de éste nuevo algoritmo, se tienen dos casos:

1. Si realmente $AB = C$, entonces toda llamada a *Verificarmultiplicación* devuelve necesariamente el valor "*verdadero*", ya que, no existe la posibilidad de seleccionar aleatoriamente un vector x tal que $xAB \neq xC$, y por lo tanto *RepetirVerificarmultiplicación* devuelve "*verdadero*" después de pasar k veces por el bucle. En éste caso la probabilidad de error es cero.

2. Si $AB \neq C$, entonces la probabilidad de que una llamada a *Verificarmultiplicación* devuelva "*verdadero*" (incorrectamente) es a lo más $\frac{1}{2}$, como se probó anteriormente. Estas posibilidades se multiplican porque las escogencias del vector x son independientes entre llamadas consecutivas, por lo tanto, la probabilidad de que k llamadas consecutivas devuelvan verdadero (incorrecto) es a lo más 2^{-k}. Además, esta es la única forma en que *RepetirVerificarmultiplicación* devuelve una respuesta incorrecta.

De lo anterior, se concluye que el algoritmo *RepetirVerificarmultiplicación* es $(1 - 2^{-k})$-correcto. Cuando $k = 10$ el algoritmo es 99,9 %-correcto, con $k = 20$ la probabilidad de error es de al menos una entre un millon [BB97]. Recuerde que para este problema si el algoritmo devuelve "falso" la respuesta es confiable, de ahí que se tenga el decrecimiento espectacularmente rápido de la probabilidad de error del algoritmo, ya que esta es una característica típica de los algoritmos de Monte Carlo que resuelven problemas de decisión en los cuales la probabilidad de error de alguna de las respuestas obtenida es cero.

El algoritmo que se ha presentado tiene un interés práctico limitado porque se necesitan $3n^2$ multiplicaciones por escalar para calcular xAB y xC, en comparación con las n^3 que se necesitan para calcular AB por el método directo. Si se necesita que la probabilidad de error no sea mayor de una en un millon y si de hecho $C = AB$, entonces las 20 ejecuciones necesarias de *Verificarmultiplicación* efectuarán $60n^2$ multiplicaciones por escalar lo cual no es mucho mejor que n^3, a no ser que n sea mayor que 60. Sin embargo, esta aproximación podría resultar de utilidad si fuese necesario verificar productos de matrices muy grandes.[BB97]

2.1.2. Verificador de composición

El algoritmo de Monte Carlo más famoso es el que decide si un número impar dado es primo o no. En este trabajo se mostrará el complemento[9] del problema anterior, es decir el problema de decidir si un número impar dado es o no compuesto.

Diferentes algoritmos probabilísticos tiempo polinomial están disponibles para el problema de verificar si un número impar dado es o no compuesto, todos ellos basados en una técnica muy simple, llamada la abundancia de testigos. La entrada a un problema muchas veces satisface una cierta propiedad, siempre que cierto objeto llamado testigo exista. Mientras que puede ser difícil encontrar tales testigos determinísticamente algunas veces es posible probar que bastantes testigos existen, permitiendo que ellos sean buscados eficientemente por simple generación aleatoria.

En el caso de los testigos de composición, la entrada es un entero n y la propiedad a verificar es, si n es compuesto. El siguiente lema podría ser una regla para los testigos de composición, Es decir los enteros a con $1 \leq a < n$ para los cuales el pequeño teorema de Fermat no se cumple, el cual se enuncia a continuación.

LEMA 1. *[PC94]Si un número $n > 2$ es compuesto, entonces un entero a existe con $1 \leq a < n$ tal que $a^{n-1} \not\equiv 1 \pmod{n}$.*

[9]Es el complemento del lenguajes, él cual se formalizará en el capítulo 3

Del lema anterior se obtiene la siguiente definición:

Definición 2.1. *Dado un entero n, un entero a es un testigo de composición para n si* $1 \leq a < n$ *y* $a^{n-1} \not\cong 1 \pmod{n}$.

Claramente, si un testigo a existe, esto es, existe un entero a para el cual no se cumple el pequeño teorema de Fermat, entonces n es compuesto. Pero la pregunta realmente importante es si n es compuesto. ¿cómo se encuentran muchos testigos de su composición?

Recuerde que, para cualquier entero n, el conjunto $\Phi(n)$ de elementos inversos módulo n se define como:

$$\Phi(n) = \{a : 1 \leq a < n \text{ y } MCD(a, n) = 1\}$$

y su cardinalidad es denotada por $\phi(n)$.
Considere el conjunto $\Phi^c(n)$, el complemento de $\Phi(n)$. Si $\Phi^c(n)$ no es vacío, entonces existe al menos un elemento $a \in \Phi^c(n)$ lo cual implica que $a^{n-1} \not\cong 1 \pmod{n}$, en efecto sea $a \in \Phi^c(n)$ y suponga que.

$$\begin{aligned} & a^{n-1} \cong 1 \pmod{n} \quad \text{entonces} \\ & = a * a^{n-2} \cong 1 \pmod{n} \quad \text{sea } b = a^{n-2} \\ & = a * b \cong 1 \pmod{n} \end{aligned}$$

De ahí que a tiene un inverso módulo n, es decir $a \in \Phi(n)$[10], lo cual es una contradicción. Por lo tanto, para todo elemento $a \in \Phi^c(n)$ se tiene que $a^{n-1} \not\cong 1 \pmod{n}$, es decir cada uno de los elementos de $\Phi^c(n)$ es un testigo de la composición de n.

Entonces se esperaría que para cualquier número compuesto n, se dé cualquiera de los siguientes casos: $\Phi^c(n)$ sea suficientemente grande o $\Phi^c(n)$ sea pequeño, pero $\Phi(n)$ sea rico en testigos; desafortunadamente, existen algunos números compuestos n para los cuales ninguno de los casos de arriba se cumplen, es decir $\Phi^c(n)$ es relativamente pequeño y ningún elemento de $\Phi(n)$ es un testigo; tales números se definen como sigue. Para cualquier entero n, sea K_n el conjunto de enteros a tales que $1 \leq a < n$ y $a^{n-1} \equiv 1$

[10]Porque, $\Phi(n)$ es un grupo sobre la multiplicación módulo n, este resultado se probará más adelante

(mód n). Un número compuesto n se dice que es un *número de Carmichael* si $K_n = \Phi(n)$

Ejemplo 5.

En la tabla 2.2 se mostrarán números compuestos, primos y números de Carmichael[11], su factorización, el número de elementos primos relativos con ellos, y en la última columna la frecuencia de los testigos de composición.

n	FACTORIZACIÓN	$\phi(n)$	PORCENTAJE
8	2.2.2	4	0.875
9	3.3	6	0.67
17	17	16	0
15	3.5	8	0.67
35	5.7	10	0.83
120	2.2.2.3.5	60	0.94
204	2.2.51	64	0.62
450	2.5.5.3.3	120	0.99
561	3.11.17	320	0.43
724	2.2.181	360	0.99
1105	5.13.17	768	0.3
1729	7.13.19	1296	0.25
2465	5.17.29	1792	0.27
2821	7.13.31	2160	0.23

tabla 2.2: Números enteros n y la frecuencia de sus testigos de composición

El siguiente teorema muestra que el concepto de testigo de composición es "bueno"para números que no son números de Carmichael. Para probar esto primero se necesita el siguiente resultado de teoría de grupos.

LEMA 2. *[PC94, FRA87] Para cualquier grupo finito G y para cualquier subgrupo propio S de G, la cardinalidad de S es un divisor de la cardinalidad de G.*

[11]Los números de Carmichael son los que se escriben en negrita.

Demostración. Suponga que la cardinalidad de G es n y que la cardinalidad de S es m. Sea R_s una relación binaria en G tal que $\langle x, y\rangle \in R_s$ si, y sólo si, $x^{-1}y \in S$, donde x^{-1} representa el inverso multiplicativo de x en G, se probará que R_s así definida es una relación de equivalencia.

- $\langle x, x\rangle \in R_s$

  En efecto, como S es un subgrupo de G, S tiene el elemento identidad e, donde $x^{-1}x = e \in S$, de ahí, $\langle x, x\rangle \in R_s$

- $\langle x, y\rangle \in R_s \longrightarrow \langle y, x\rangle \in R_s$

  $\langle x, y\rangle \in R_s \Longleftrightarrow x^{-1}y \in S$, como S es un subgrupo, el inverso de $x^{-1}y$ existe en S, es decir, el elemento $(x^{-1}y)^{-1} \in S$, entonces

$$(x^{-1}y)^{-1} \in S \Longleftrightarrow (x^{-1})^{-1}y^{-1} \in S \Longleftrightarrow xy^{-1} \in S$$
$$\Longleftrightarrow y^{-1}x \in S \Longleftrightarrow \langle y, x\rangle \in R_s$$

- $\langle x, y\rangle \in R_s \wedge \langle y, z\rangle \in R_s \Longrightarrow \langle x, z\rangle \in R_s$

  Por hipótesis, $\langle x, y\rangle \in R_s$ y $\langle y, z\rangle \in R_s \Longleftrightarrow x^{-1}y \in S \wedge y^{-1}z \in S$. Claramente el producto de elementos de S están en S, por ser S un subgrupo, entonces,

$$(x^{-1}y)(y^{-1}z) = x^{-1}(yy^{-1})z = x^{-1}ez$$
$$= x^{-1}z \in S \Longleftrightarrow \langle x, z\rangle \in R_s$$

Así, R_s es una relación de equivalencia y la clase de equivalencia $\bar{x}$, que contiene a x, se calcula fácilmente como:

$$\bar{x} = \{y \in G : y^{-1}x \in S\} = \{y \in G : y^{-1}x = s, s \in S\}$$
$$= \{y \in G : x = sy \; para \; s \in S\} = xS$$

las cuales son precisamente las clases laterales izquierdas de S; se mostrará que cualesquiera dos clases laterales izquierdas tienen el mismo número de elementos. Considere

la transformación $f_x : S \longrightarrow xS$ dada por $(s)f_x = xs$. f_x así definida es una función sobreyectiva; en efecto si $s_1 \, y \, s_2 \in S$ y $xs_1 = xs_2$ entonces $s_1 = s_2$ por la ley cancelativa del grupo, así,f_x lleva a S de manera uno a uno sobre xS. De ahí que toda clase lateral izquierda tiene el mismo número de elementos que S, esto es, $|[x]_s| = m$ para algún $x \in S$, donde $[x]_s$ denota la clase lateral izquierda determinada por x

De lo anterior, si hay r clases laterales izquierdas debemos tener que $n = rm$, esto es, la cardinalidad de S divide a la cardinalidad de G. $\diamondsuit$

Para mayor claridad en la demostración del teorema que se enunciará a continuación se mostrarán algunos resultados de teoría de anillos y grupos, las demostraciones serán omitidas porque ellas no corresponden al área de la complejidad , que es el área de estudio en este documento.

Definición 2.2. *Si a y b son dos elementos distintos de cero de un anillo*[12] *R tal que $ab = 0$, entonces a y b son divisores de cero.*

Por ejemplo en $\mathbf{Z}_{12}$[13] los elementos $\{2, 3, 4, 6, 8, 9, 10\}$ son divisores de cero, note que son precisamente los elementos de $\mathbf{Z}_{12}$ que no son primos relativos con 12.

Teorema 2.1.1. *[FRA87]En el anillo $\mathbf{Z}_n$ los divisores de cero son precisamente aquellos elementos que no son primos relativos con n.*

Con todo lo anterior se puede probar el siguiente teorema.

TEOREMA 2.1. *[PC94]Si n es un número compuesto que no es de Carmichael, entonces el número de testigos de la composición de n es al menos $\frac{\phi(n)}{2}$.*

Demostración. Según lo anterior el conjunto $\Phi(n)$ es igual al conjunto de los $a \in \mathbf{Z}_n$, $a \neq 0$ tales que a no es divisor de cero. Se Probará que éste conjunto es un grupo sobre la multiplicación módulo n.

[12]Un **anillo**$\langle R, +, . \rangle$ es un conjunto R junto con dos operaciones binarias + y $*$, que llamaremos suma y multiplicación, definidas en R tales que (1) $\langle R, + \rangle$ es un grupo abeliano; (2) La multiplicación es asociativa; (3)Para toda $a, b, c \in R$ se cumple la ley distributiva izquierda y derecha.

[13]$\mathbf{Z}_n$ es el conjunto de residuos módulo n; $\mathbf{Z}_n$ es un anillo para cualquier n

- $\Phi(n)$ es cerrado. Sean a y $b \in \Phi(n)$, si $ab \notin \Phi(n)$, entonces existe $c \neq 0 \in \mathbf{Z}_n$ tal que $(ab)c = 0$, por lo cual $a(bc) = 0$. Ahora como $b \in \Phi(n)$ y $c \neq o$ se tiene que a es un divisor de cero y así, $a \notin \Phi(n)$, lo cual contradice la hipótesis.

- La multiplicación módulo n es asociativa y además $1 \in \Phi(n)$.

- Para todo $a \in \Phi(n)$ existe b tal que $ab = 1$.

  Sean $a, aa_1, aa_2, \ldots, aa_r$ los elementos de $\Phi(n)$, los cuales son todos distintos porque si $aa_i = aa_j$, entonces $a(a_i - a_j) = 0$ y como $a \neq 0$, entonces $a_i - a_j = 0$, lo cual implica que $a_i = a_j$, luego se tendría que $a1 = 1$ o alguna aa_i debe ser 1. Y así, a tiene inverso multiplicativo.

Por lo anterior, $\Phi(n)$ es un grupo sobre la multiplicación módulo n. Ahora se mostrará que K_n es un subgrupo propio de $\Phi(n)$.

- Sea $a \in K_n$, entonces $a^{n-1} \cong 1 \pmod{n}$, de ahí que n divide a $a^{n-1} - 1$, entonces existe $k \in \mathbb{Z}$ tal que $nk = a^{n-1} - 1 \Rightarrow 1 = a^{n-1} - nk$, esto es, 1 es el menor entero que es combinación lineal de a^{n-1} y n, luego, $MCD(a^{n-1}, n) = 1$. De la desigualdad $1 = a^{n-1} - nk$ se tiene que $1 = a(a^{n-2}) - nk$, sea $m = a^{n-2}$, entonces $1 = am - nk$, lo cual es equivalente a que $MCD(a, n) = 1$, entonces $a \in \Phi(n)$, esto es, K_n es subconjunto propio de $\Phi(n)$ ya que existen elementos en $\Phi(n)$ que no están en K_n[14]

- la relación de congruencia es cerrada bajo la multiplicación módulo n y la identidad $1 \in K_n$.

- Para toda $a \in K_n$, $a^{-1} \in K_n$, se prueba de manera análoga como se probó el inverso para el grupo.

Por lo tanto K_n es subgrupo propio de $\Phi(n)$, entonces del lema anterior se tendría que $|K_n|$ es un divisor de $\phi(n)$, esto es existe $t \in \mathbb{Z}$ tal que $|K_n|t = \phi(n)$ y como K_n es subgrupo propio de $\Phi(n)$, entonces $t \geq 2$ de ahí que, $|K_n| < \frac{\phi(n)}{2}$.

[14]Por ejemplo para $n = 8$; $K_n = \{1\}$ y $\Phi(n) = \{1, 3, 5, 7\}$

De lo anterior, se tiene que el número de elementos $1 \leq a < n$ tales que $a^{n-1} \cong (1 mod n)$ es a lo más $\frac{\phi(n)}{2}$. De ahí que los $a \in [1, n)$ tales que $a^{n-1} \ncong (1 mod n)$ es al menos $\frac{\phi(n)}{2}$, esto es, los testigos de la composición de n son al menos $\frac{\phi(n)}{2}$ $\diamondsuit$

Faltaría tratar el caso en el cual n es un número de Carmichael; para hacer esto, es necesario modificar la definición de testigo de composición como sigue:

Definición 2.3. *Dado un entero n, un entero a es un testigo de composición para n si las siguientes dos condiciones son verdaderas:*

1. $1 \leq a < n$
2. a) $a^{n-1} \ncong 1(mod n)$ *ó*

   b) *existe un entero i tal que 2^i divide a $n-1$ y $1 < MCM((a^{\frac{(n-1)}{2}} - 1), n) < n$*

Es claro que si un testigo a existe entonces n es compuesto, además, puesto que esta definición incluye la definición anterior de testigo de composición, el teorema anterior puede ser aplicado. Para probar un resultado similar para los números de Carmichael, primero es necesario mostrar los siguientes lemas

LEMA 3. *[PC94] Para cualquier número primo p y para cualquier entero $k \geq 1$, $\phi(p^k) = p^{k-1}(p-1)$.*

Demostración. Primero se probará que para cualquier número entero a con $1 < a < p^k$, $MCD(a, p^k) = 1$ si, y sólo si, p no divide a a.

$\Rightarrow$) Como $MCD(a, p^k) = 1$, entonces existen k_1, $k_2 \in \mathbb{Z}$ $k_1 \neq 0, k_2 \neq 0$ tales que:

$$1 = k_1 a + k_2 p^k \Rightarrow 1 = k_1 a + (k_2 p^{k-1})p$$

y haciendo

$$k_3 = k_2 p^{k-1} \Rightarrow 1 = k_1 a + k_3 p$$

Como 1 es el menor entero que se puede escribir como combinación lineal de a y p, entonces $MCD(a, p) = 1$, de ahí claramente se tiene que p no divide a a.

$\Leftarrow$) Suponga que p no divide a a y que $MCD(a, p^k) \neq 1$, entonces $MCD(a, p^k) = d$, para $d \neq 1, d \in \mathbb{Z}$, entonces existen enteros $k_1, k_2 \in \mathbb{Z}$ tal que

$$d = k_1 a + k_2 p^k \Rightarrow d = k_1 a + k_3 p$$

donde $k_3 = k_2 p^{k-1}$, entonces $MCD(a, p) = d$, pero como p es primo y $d \neq 1$ entonces $d = p$, de ahí que p divide a a, lo cual es una contradicción porque por hipótesis p no divide a a, así el $MCD(a, p^k) = 1$.
Considere el conjunto formado por los múltiplos de p menores que p^k, es decir el conjunto $\{p, 2p, \ldots, p^k - p\}$. En este conjunto se tienen $p^{k-1} - 1$ elementos. Incluyendo el número 1 se tienen p^{k-1}, luego $\phi(p^k) = p^k - p^{k-1} = p^{k-1}(p-1)$ $\diamondsuit$

Antes de presentar los siguientes teoremas enunciemos algunos resultados que serán de gran utilidad.

Definición 2.4. *Sean a y n enteros positivos, entonces el menor entero positivo x tal que $a^x \cong 1 \pmod{n}$ es llamado el orden de a módulo n, y se nota $Ord_n a$.*

El siguiente teorema se utiliza para encontrar las soluciones de la ecuación $a^x \cong 1 \pmod{n}$.

Teorema 2.1.2. *[KEN97] Si a y b son primos relativos con $n > 0$, entonces $a^x \cong 1 \pmod{n}$, si y sólo si, el orden de a módulo n divide a x.*

Definición 2.5. *Si r y n son primos relativos, con $n > 0$, y si $ord_n r = \phi(n)$, entonces r es llamada una raíz primitiva módulo n.*

Definición 2.6. *Un exponente universal de un entero positivo n es un entero u tal que $a^u \cong 1 \pmod{n}$.*

Definición 2.7. *El menor exponente universal de un entero positivo n es llamado el mínimo exponente universal de n y se denota $\lambda(n)$.*

Teorema 2.1.3. *[KEN97] Sea n un entero positivo con factorización en potencias de primos $n = 2^{t_0} p_1^{t_1} \ldots p_m^{tm}$, entonces $\lambda(n)$ (el mínimo exponente universal de n) está dado por*

$$\lambda(n) = mcm(2^{t_0}, \phi(p_1^{t_1}), \ldots, \phi(p_m^{tm}))$$

Teorema 2.1.4. *[PC94] Si n es un número de Carmichael impar, entonces n es el producto de r factores primos impares diferentes $p_1, p_2, \ldots, p_r$ con $r \geq 3$.*

Demostración. Para cualquier número n impar $n = p_1^{k_1} p_2^{k_2} \ldots p_r^{k_r}$ sea

$$\lambda(n) = mcm(\phi(p_1^{k_1}), \phi(p_2^{k_2}), \ldots \phi(p_r^{k_r}))$$

donde mcm representa la función mínimo común múltiplo, entonces para cualquier $a \in \Phi(n)$, $a^{\lambda(n)} \cong 1 \pmod{n}$, en efecto.

- si $r = 1$, entonces $a^{\lambda(n)} = a^{\Phi(n)}$ y del teorema de Euler[15] [FRA87] se tiene que $a^{\Phi(n)} \cong 1 \pmod{n}$, de ahí, $a^{\lambda(n)} \cong 1 \pmod{n}$.

- por otro lado, sea a un entero primo relativo con n, entonces por el teorema de Euler se tiene que $a^{\phi(p^k)} \cong 1 \pmod{p^k}$, donde p^k es una potencia de un primo en la factorización de n; puesto que $\lambda(n)$ es un múltiplo de cada $\phi(p_i^{k_i})$ con $1 \leq i \leq r$, se tiene que $\phi(p_i^{k_i}) | \lambda(n)$ para $i = 1, 2 \ldots, r$. Además x es la solución de la congruencia $a^x \cong 1 \pmod{n}$ si, y sólo si, $ord_n\ a | x$. Luego $\lambda(n)$ es la solución de $a^{\lambda(n)} \cong 1 \pmod{p_i^{k_i}}$ para $i = 1, 2, \ldots, r$, entonces $a^{\lambda(n)} \cong 1 \pmod{n}$, por corolario[16].

  Ahora se probará que para cualquier $a \in \Phi(n)$, $\lambda(n)$ es el menor exponente tal que $a^{\lambda(n)} \cong 1 \pmod{n}$. Como n es un número impar de Carmichael, aplicando el teorema (2.1.2), se tiene que el mínimo exponente universal de n está dado por:

  $$\lambda(n) = mcm(\phi(p_1^{t_1}), \phi(p_2^{t_2}), \ldots, \phi(p_r^{t_r}))$$

  Esto es, $\lambda(n)$ es el menor exponente tal que para cualquier $a \in \Phi(n)$, $a^{\lambda(n)} \cong 1 \pmod{n}$.

  Se mostrará que n es un número de Carmichael, si y sólo si, $\lambda(n)$ es un factor de $n-1$.

[15] Si a es un entero primo relativo con n, entonces $a^{\phi(n)} - 1$ es divisible entre n, esto es $a^{\phi(n)} \cong 1 \pmod{n}$.

[16] **Corolario** si $a \cong b \pmod{m_1}$, $a \cong b \pmod{m_2}$, ..., $a \cong b \pmod{m_k}$ donde a y b son enteros y $m_1 m_2 \ldots m_r$ son potencias de primos relativos, entonces, $a \cong b \pmod{m_1 m_2 \ldots m_r}$.

$\Rightarrow$) Si n es un número de Carmichael, entonces, $a^{n-1} \cong 1 \pmod{n}$, para todo entero positivo a tal que $MCD(a,n) = 1$, como $a^{\lambda(n)} \cong 1 \pmod{n}$ y además $\lambda(n)$ es el menor entero que cumple la congruencia, entonces $\lambda(n)|n-1$, lo cual implica que existe $k \in \mathbb{Z}$ tal que $k\lambda(n) = n-1$, esto es $\lambda(n)$ es un factor de $n-1$.

$\Leftarrow$) Como $a^{\lambda(n)} \cong 1 \pmod{n}$, entonces $a^{k\lambda(n)} \cong 1^k \pmod{n}$, si y sólo si, $a^{n-1} \cong 1 \pmod{n}$, entonces n es un número de Charmichael.

Así, se tiene que el $MCD(n, \lambda(n)) = 1$, puesto que n no tiene un factor primo repetido ya que en éste caso podría ser un factor de n y $\lambda(n)$, además n no puede ser producto de dos números primos. En efecto, sea $n = pq$ con $p < q$ y p, q primos relativos, entonces $\lambda(n)$ es un factor de $n-1$, es decir, es un factor de $pq-1$. También, $\lambda(n) = mcm(\phi(p), \phi(q)) = mcm(p-1, q-1)$. Por lo tanto $\lambda(n)$ es múltiplo de $q-1$, el cual podría dividir a $pq-1$, pero.

$$\frac{pq-1}{q-1} = \frac{pq\ -p\ +\ p\ -\ 1}{q-1} = p\ +\ \frac{p-1}{q-1}$$

Y puesto que $\frac{p-1}{q-1}$ no es un entero, no es posible que $q-1$ divida a $pq-1$. Así, n debe ser el producto de más de tres factores primos diferentes.

$\diamondsuit$

TEOREMA 2.2. *[PC94] Si n es un número impar de Carmichael, entonces el número de testigos de composición de n es al menos tres cuartos de $\phi(n)$.*

Demostración. Sean p_1, p_2, p_3 tres factores primos diferentes de n. Se probará que $p_i - 1$ divide a $n-1$. Por teorema 2.1.2 existe un entero a con $Ord_n a = \lambda(n)$, donde $\lambda(n)$ es el mínimo exponente universal (definición 2.1.2) y como $a^{n-1} \cong 1 \pmod{n}$, la definición 2.4 dice que $\lambda(n)|n-1$. Además $\lambda(n)$ es un múltiplo de cada $\phi(p_i)$, $i = 1, 2, 3$. De ahí,

$$\phi(p_i)|\lambda(n) \ \Leftrightarrow\ p_i - 1|\lambda(n)$$

Cómo $p_i - 1|\lambda(n)$ y $\lambda(n)|n-1$, entonces, $p_i - 1$ divide a $n-1$

Se denotará por e_i al más grande de los j tales que 2^j divide a $p_i - 1$ y se asumirá que $e_1 = e_2 = e_3 = e$ (se hace un análisis similar en caso de $e_1 = e_2 < e_3$ y $e_1 < e_2 \leq e_3$), puesto que, $p_i - 1 = 2^e m_i$ donde m_i es un entero impar y como $p_i - 1$ divide a $n - 1$ se sigue que existe $t \in \mathbb{Z}$ tal que

$$n - 1 = t(p_i - 1) = t(2^e m_i)$$

Como $n - 1$ es par también puede ser dividido por otra potencia de 2, luego t es de la forma $t = 2^k t'$ donde $k \in \mathbb{Z}^+ \cup \{0\}$ y t' es un entero impar, de lo anterior se tiene que.

$$\begin{aligned} n - 1 &= 2^k t' 2^e m_i \\ &= 2^k 2^e (t' m_i) = 2^e 2^k m \end{aligned}$$

Así, $n - 1 = 2^e 2^k m$, donde m es un número impar divisible por m_i y $k \in \mathbb{Z}^+ \cup \{0\}$.

Sea $d = \dfrac{n-1}{2^{k+1}} = \dfrac{2^e 2^k m}{2^{k+1}} = \dfrac{2^e m}{2}$, claramente $p_i - 1$ no divide a d,puesto que.

$$\frac{d}{p_i - 1} = \frac{\frac{2^e m}{2}}{2^e m_i} = \frac{2^e m}{2 * 2^e m_i} = \frac{m}{2m_i} \qquad \text{pero, } m = t' m_i$$

$$\Rightarrow \frac{m}{2m_i} = \frac{t' m_i}{2m_i} = \frac{t'}{2}$$

lo cual no es un número entero porque t' es un entero impar; además $\frac{p_i - 1}{2}$ divide a d ya que.

$$\frac{d}{\frac{p_i - 1}{2}} = \frac{\frac{2^e m}{2}}{\frac{2^e m_i}{2}} = \frac{2^e m}{2^e m_i} = \frac{m}{m_i}$$

pero, $m = t' m_i$ entonces

$$\frac{m}{m_i} = \frac{t' m_i}{m_i} = t' \text{ lo cual es un entero impar.}$$

Sea b_i una raíz primitiva módulo p, esto es, para cualquier entero t, $b_i^t \cong 1 \pmod{p}$ si y sólo si, $p-1$ divide a t. Se Probará que las potencias $b_i, b_i^2, \ldots, b_i^{p_i-1}$ son congruentes módulo p_i a los números $1, 2, \ldots, p_i - 1$.

Para garantizar lo anterior sólo es necesario mostrar que las potencias de b_i son todas primos relativos con p_i. Puesto que $MCD(b_i, p_i) = 1$, entonces $MCD(b_i^k, p_i) = 1$ para cualquier entero positivo k. Éste resultado se probará por inducción sobre k.

- Para $k = 2$

  Por hipótesis se tiene que $MCD(b_i, p_i) = 1$ lo cual es equivalente a que 1 es el menor entero que se puede escribir como combinación lineal de b_i y p_i [KEN97], esto es existen enteros t_1 y t_2 tales que.

$$\begin{aligned} 1 = b_i t_1 + p_i t_2 \Leftrightarrow 1 &= (b_i t_1 + p_i t_2)(b_i t_1 + p_i t_2) \\ \Leftrightarrow 1 &= b_i^2 t_1^2 + b_i t_1 p_i t_2 + p_i t_2 b_i t_1 + p_i t_2 p_i t_2 \\ \Leftrightarrow 1 &= b_i^2 t_1^2 + p_i(b_i t_1 t_2 + t_2 b_i t_1 + p_i t_2^2) \\ \Leftrightarrow 1 &= b_i^2 t_3 + p_i t_4 \end{aligned}$$

  Como $t_1, t_2\, y\, b_i \in \mathbb{Z}$, entonces, $t_1^2 = t_3 \in \mathbb{Z}$ y $b_i t_1 t_2 + t_2 b_i t_1 + p_i t_2^2 = t_4 \in \mathbb{Z}$, Por lo tanto $MCD(b_i^2, p_i) = 1$.

- Suponga que el resultado es válido para $k-1$, es decir si $MCD(b_i, p_i) = 1$, entonces $MCD(b_i^{k-1}, p_i) = 1$ y se probará que se cumple para k.

  Por la hipótesis inductiva, se tiene que existen enteros t_1, t_2, t', t'' tales que $1 = b_i t_1 + p_i t_2$ y $1 = b_i^{k-1} t' + p_i t''$, multiplicando estos dos resultados se tiene que.

$$\begin{aligned} 1 =& (b_i t_1 + p_i t_2)(b_i^{k-1} t' + p_i t'') \\ &= b_i b_i^{k-1} t' t_1 + b_i^{k-1} t' p_i t_2 + p_i t'' b_i t_1 + p_i t'' p_i t_2 \\ &= b_i^k t' t_1 + p_i(b_i^{k-1} t' t_2 + t'' b_i t_1 + t'' p_i t_2) \end{aligned}$$

  Como se tiene que $t't_1$, y, $b_i^{k-1} t' t_2 + t'' b_i t_1 + t'' p_i t_2$ son enteros, se llamarán x, y, y, respectivamente, entonces se tiene que.

$$1 = b_i^k x + p_i y \Leftrightarrow MCD(b_i^k, p_i) = 1$$

Puesto que estas potencias son todas primas relativas entonces, $b_i^j \cong b_i^k \pmod{p_i}$, donde $j, k \in \mathbb{Z}^+$ si, y sólo si, $j \cong k \pmod{p_i}$ por teorema [KEN97, JGR99].

Además, para $1 \leq j \leq p_i - 1$ y $1 \leq k \leq p_i - 1$ la congruencia $j \cong k \pmod{p_i}$ implica que $j = k$, porque, las dos potencias son equivalentes módulo p_i, así se ha mostrado que las primeras $p_i - 1$ potencias de b_i son equivalentes módulo p_i a los números $1, 2, \ldots, p_i - 1$. De ahí, para cada $a \in \Phi(n)$, un correspondiente r_i debe existir con $1 \leq r_i \leq p_i - 1$ tal que $a \cong b_i^{r_i} \pmod{p_i}$.

Note que si r_i es par, entonces p_i divide a $a^d - 1$, en efecto, $r_i = 2r_i'$, $r_i' \in \mathbb{Z}$, entonces como $a \cong b^{r_i} \pmod{p_i}$ elevando a ambos lados de la desigualdad por d.

$$a^d \cong b_i^{r_i d} \pmod{p_i} = b_i^{2r_i' d} \pmod{p_i}$$

Pero, $\dfrac{p_i - 1}{2}$ divide a d, entonces existe $t \in \mathbb{Z}$ tal que $d = t(\frac{p_i - 1}{2})$, entonces, $2d = t(p_i - 1)$, esto es $p_i - 1$ divide a $2d$. Además $b^{p_i - 1} \cong 1 \pmod{p_i}$ y $a^d \cong b_i^{2r_i' p_i - 1} \pmod{p_i}$, entonces $a^d \cong 1 \pmod{p_i}$. De ahí, p_i divide a $a^d - 1$.

Si r_i es impar, entonces p_i no divide a $a^d - 1$, en efecto,

$$\begin{aligned} a^d \cong b_i^{r_i d} \pmod{p_i} \qquad & \text{pero, } p_i - 1 \text{ no divide a } d, \text{ entonces} \\ a^d \ncong 1 \pmod{p_i} & \end{aligned}$$

De ahí, claramente p_i no divide a $a^d - 1$

Luego, para cualquier $a \in \Phi(n)$ tal que la correspondiente terna $\langle r_1, r_2, r_3 \rangle$ contiene al menos un número par y al menos un número impar, tenemos que a es un testigo, esto es, $1 < MCD(a^d - 1, n) < n$. Si por ejemplo r_1 es par y r_2 es impar, entonces p_1 divide a ambos $a^d - 1$ y n, y p_2 no divide a $a^d - 1$. Además $1 < MCD(a^d - 1, n) < n$.

Así para cualquier $a \in \Phi(n)$ tal que la correspondiente tripla contenga al menos un número

par y al menos un número impar, se tiene que a es un testigo, esto es existe un entero $k+1$ tal que 2^{k+1} divide $n-1$ y $1 < MCD(a^{\frac{n-1}{2^{k+1}}} - 1, n) < n$; el número $k+1$ existe, ya que $n-1 = 2^k 2^e m$, $k \geq 0$. Suponga por ejemplo que r_1 es par y que r_2 es impar. Cómo r_1 es par entonces p_1 divide a $a^{\frac{n-1}{2^{k+1}}} - 1 = a^d - 1$ y p_1 divide a d, entonces el $MCD(a^d - 1, n) = p_1$ y p_2 no divide a $a^d - 1$, luego $1 < MCD(a^d - 1, n) < n$.

Ahora para cada r_i, $i = 1, 2, 3$ se tiene dos posibilidades para las paridades, es decir, r_i puede ser par o impar; luego para $\langle r_1, r_2, r_3 \rangle$ se tiene 8 posibles paridades y como de estas se deben escoger aquellas que contengan al menos un número par y al menos un número impar; en la tabla 2.3 se mostrarán las ocho posibles paridades de $\langle r_1, r_2, r_3 \rangle$ y se resaltarán aquella posibilidades para las cuales un elemento $a \in \Phi(n)$ es un testigo.

paridades
$\langle par, impar, par \rangle$
$\langle impar, impar, par \rangle$
$\langle impar, par, par \rangle$
$\langle impar, par, impar \rangle$
$\langle impar, impar, impar \rangle$
$\langle par, par, par \rangle$
$\langle par, impar, impar \rangle$
$\langle par, par, impar \rangle$

tabla 2.3: Paridades para la terna $\langle r_1, r_2, r_3 \rangle$

En la tabla 2.3 se ven que para cada $a \in \Phi(n)$ hay 6 posibilidades de escoger r_1, r_2, r_3 con al menos un número par y al menos un número impar. Luego el número de testigos de la composición de n será: $\frac{6\phi(n)}{8} = \frac{3\phi(n)}{4}$ si $n = p_1 p_2 p_3$.

Como $n = p_1 p_2 ... p_r$ con $r \geq 3$, se verá para el caso $r = 4$ cuantos testigos de composición

existen. Para las paridades de $\langle r_1, r_2, r_3, r_4\rangle$ se tienen 16 posibilidades y para cada $a \in \Phi(n)$ hay 14 posibilidades de escoger $\langle r_1, r_2, r_3, r_4\rangle$ con al menos un número par y un número impar. Luego el número de testigos de la composición de n sería

$$\frac{14\Phi(n)}{16} = \frac{7\Phi(n)}{8}$$

En general, el número de testigos para la composición de n si $n = p_1p_2 \ldots P_r$, con $r \geq 3$ es $\frac{(2^n-2)\Phi(n)}{2^n}$. Pero, $\frac{2^n-2}{2^n} > \frac{3}{4}$ por que n es un número de Carmichael impar.

Por lo tanto, el número de testigos de la composición de n es el menos $\dfrac{3\phi(n)}{4}$ $\diamondsuit$

Combinando los teoremas 2.2 y 2.1 se tiene el siguiente resultado.

Corolario 2.2. *[PC94] Si n es un número compuesto impar, entonces éste admite al menos $\frac{(n-1)}{2}$ testigos de composición.*

Consideremos el siguiente algoritmo probabilístico.

Inicio {Entrada: $n > 2$}
si n es par **entonces** aceptar
sino
inicio
$a \longleftarrow random(1, n-1)$
si a es testigo de composición de n **entonces** aceptar
sino rechazar
fin
fin.

Chequear si a es un testigo de composición de un entero n lo hace en tiempo polinomial, ya que sólo necesitaría verificar que alguna de las dos condiciones de la definición 2.3 se cumple. Además, del corolario de arriba se sigue que el error de probabilidad del algoritmo es menor que $\frac{1}{2}$, esto es si el algoritmo acepta, entonces se tiene con seguridad

de que el número es compuesto, mientras que si rechaza, entonces se puede establecer que n es primo con probabilidad de al menos $\frac{1}{2}$, el algoritmo anterior es un algoritmo de Monte Carlo, este algoritmo es eficiente pero algunas veces retorna respuestas incorrectas, aunque la probabilidad de error se puede hacer bastante pequeña permitiendo más tiempo de ejecución al algoritmo.

2.2. ALGORITMOS DE LAS VEGAS

Un algoritmo tipo ***Las Vegas*** siempre retorna una respuesta correcta pero no garantiza el tiempo total de ejecución, aunque con alta probabilidad éste será bajo. Estos algoritmos toman decisiones al azar como ayuda para encontrar la solución correcta antes que un algoritmo determinista; a diferencia de los algoritmos de Monte Carlo (sección 2.1) nunca proporcionan una respuesta incorrecta; si no encuentran la solución, lo admiten.

Intuitivamente, un algoritmo de las Vegas es un algoritmo probabilístico que siempre retorna respuestas correctas y cuando no esta seguro lo acepta, es decir en este caso los algoritmos de las Vegas devuelven la respuesta "no sé".

Estos algoritmos son clasificados en dos tipos dependiendo de las posibilidades de encontrar la solución.

1. Los que siempre encuentran una solución correcta aunque las decisiones tomadas al azar no sean afortunadas y la eficiencia disminuya.

2. Los que debido a las decisiones desafortunadas no encuentran la solución en esa ejecución del algoritmo, esto es, pueden permitirse tomar caminos equivocados que los lleva a un callejón sin salida, haciendo imposible hallar la solución al problema en esta ejecución del algoritmos.

Los algoritmos de las Vegas de la primera clase son utilizados para problemas en los cuales el algoritmo determinista que lo resuelve es mucho más rápido en el caso promedio que en

el peor de los casos, por ejemplo el algoritmo *Quicksort*(ordenación rápida)[17]. Al permitir que estos algoritmos usen aleatoriedad para orientar su búsqueda, permite reducir y a veces eliminar la diferencia entre los casos buenos y malos.

Tenga en cuenta que, el análisis de la eficiencia del caso promedio de un algoritmo puede a veces producir resultados que lleven a una confusión, la razón es que todo análisis del caso promedio debe estar basado en una hipótesis acerca del la distribución de probabilidad de los casos que haya que manejar. Una hipótesis que sea correcta para una aplicación dada del algoritmo puede resultar desastrosamente incorrecta para una aplicación diferente [BB97], por ejemplo:

Para determinar el tiempo promedio que requiere el algoritmo Quikcsort (ordenación rápida) para ordenar n elementos, se supone que los elementos son diferentes y que tienen la misma probabilidad de ser elegidos. Sin embargo, está suposición puede no ser válida (puede ser totalmente equivocada) para algunas entradas; por ejemplo si el vector que hay que ordenar ya está casi totalmente ordenado. En general, estos algoritmos deterministas son vulnerables a una distribución de probabilidad inesperada de los casos (entradas) que puedan pasársele para resolver. Los algoritmos de las vegas no permiten estas situaciones, igualando el tiempo necesario para los diferentes casos de un tamaño dado.

El rendimiento medio de esta clase de algoritmos de las Vegas no es mejor que el algoritmo determinista correspondiente. Con alta probabilidad, los casos que requieren mucho más tiempo con el método determinista se resuelven ahora mucho más rápido, pero los casos en que el algoritmo determinista fuera especialmente bueno se llevan a un valor medio en el algoritmo de las Vegas (esto es lo que se llama efecto *Robin Hood*, "robar"tiempo a los ejemplares "ricos"para dárselo a los "pobres ").

La otra clase de algoritmos de las Vegas se caracteriza por tomar de vez en cuando de-

[17]Algoritmo de ordenación inventado por Hoare, el cual ordena n elementos en un tiempo promedio de $O(n\log(n))$ y en el peor de los casos, es decir si todos los elementos que hay que ordenar son casi iguales, el algoritmo requiere un tiempo cuadrático($\Omega(n^2)$)

cisiones que los llevan a un callejón sin salida, en este caso los algoritmos son capaces de reconocer su situación en cuyo caso se limitarán a admitir el fallo; como un algoritmo de las Vegas es un algoritmo probabilístico se tiene que la admisión de fallo es aceptable siempre y cuando esta se produzca con baja probabilidad; cuando se obtiene un fallo basta con volver a ejecutar el algoritmo sobre la misma entrada para tener una nueva posibilidad de éxito; este tipo de algoritmos son eficientes aunque den respuestas equivocadas para resolver problemas de los cuales no se conoce un algoritmo determinístico eficiente que lo resuelva. Sin embargo, no se puede establecer una cota superior para el tiempo que es necesario para encontrar una respuesta correcta si se vuelve a ejecutar el algoritmo siempre que este falle, este tiempo es razonable(polinomial) con una alta probabilidad. Un algoritmo de las Vegas deberá tener un buen rendimiento esperado sea cual sea el caso que haya que resolver.

En esta sección serán mostrados dos ejemplos de algoritmos de las Vegas. El primero es el problema de ordenar un vector V de n elementos **ordenación probabilísta** y el segundo el **problema de factorización de números grandes**, el cual es un ejemplo sofisticado de algoritmo de la Vegas que puede fallar en ocasiones, pero para el cual hay una estrategia mejor que reiniciar todo el algoritmo en caso de fallar.

2.2.1. Ordenación probabilísta

Se analizará el algoritmo Quicksort(ordenación rápida) basado en la técnica divide y venceras:

Divide: El vector de entrada $A[p..r]$ es particionado es dos subvectores no vacíos $A[p..q]$ y $A[q+1..r]$ tal que cada elemento de $A[p..q]$ es menor o igual a cada elemento de $A[q+1..r]$. El indice q es calculado por este procedimiento.

Venceras: los dos subvectores $A[p..q]$ y $A[q+1..r]$ son ordenados por llamadas recursivas a Quicksort.

Combina: Como cada subvector ya está ordenado, no necesitamos trabajo para combinarlos, el vector $A[p..r]$ ahora está ordenado.

Los algoritmos son:

Quicksort(a,p,r)

si $p < r$ **entonces**

inicio

$q \leftarrow$ partición(A, p, r)

$Quicksort(A, p, q)$

$Quicksort(A, q + 1, r)$

fin.

Para ordenar un vector A completamente, la llamada inicial a Quicksort es $Quicksort(A, 1, longitud(A))$; la clave del algoritmo anterior es el procedimiento **partición**, el cual organiza el vector $A[p..r]$ por partes.

Partición(A, p, r)

inicio

$x \leftarrow A[p]$

$i \leftarrow p - 1$

$j \leftarrow r + 1$

mientras $i \leq j$ **hacer**

inicio

repetir $j \leftarrow j - 1$ **hasta**$A[j] \leq x$

repetir $i \leftarrow i + 1$ **hasta**$A[i] \geq x$

si $i < j$ **entonces**

intercambiar $A[i] \leftarrow A[i]$

else devolver j

fin

fin.

Ejemplo 6.

Para el ejemplo se aplicará partición al siguiente vector A

3	1	4	1	5	9	2	6	5	3	5	8	9

La llamada inicial para particionar el vector A es **Partición**$(A, 1, 13)$

- En la figura 2.3(a) vemos el vector de entrada con los valores iniciales de i y j por fuera del vector a particionar para el pivote $A[1] = 4$
- En la figura 2.3(b) tenemos las posiciones de i y j verificando si $i < j$, en la primera entrada al ciclo mientras.
- En la figura 2.3(c) intercambiamos los elementos $A[i]$ y $A[j]$, es decir, $A[1]$ y $A[10]$
- En la figura 2.3(d) Podemos ver las posiciones de i y j en la segunda entrada al ciclo mientras.
- En la figura 2.3(e) intercambiamos los elementos $A[i]$ y $A[j]$, es decir, $A[5]$ y $A[7]$
- En la figura 2.3(f) Tenemos las posiciones de i y j en la última iteración del ciclo mientras. El procedimiento termina por que $j \geq i$, el valor $q = j = 5$ es devuelto; en el subvector $A[p..q]$ los elementos son menores que el pivote, mientras que en el vector $A[q + 1..r]$ los elementos son mayores o iguales al pivote=4.

El tiempo de ejecución de Quicksort depende de si la partición es balanceada o no lo es. Esto es, depende de los elementos a particionar; si la partición es balanceada el algoritmo corre muy rápido, pero si la partición no esta balanceada el algoritmo es "lento". El peor caso de Quicksort se produce cuando el procedimiento partición produce una región con $n - 1$ elementos y otra solamente con 1 elemento. Así la partición es máximamente desbalanceada, entonces cada paso recursivo del algoritmo corre en $\Theta(n^2)$; el peor caso de Quicksort ocurre cuando el vector de entrada esta completamente ordenado

El mejor de los casos de Quicksort es cuando el procedimiento partición produce dos regiones de tamaños $\frac{(n-1)}{2}$, en este caso el algoritmo tiene un tiempo de complejidad de $\Theta(n \log n)$ porque este será la solución de la recurrencia[18] que se obtiene en este caso,

[18]Cuando un algoritmo contiene un llamado recursivo a si mismo, su tiempo de ejecución puede describirse muchas veces como una recurrencia. Recordemos que una recurrencia es una igualdad o una desigualdad que describe una función en términos de sus valores sobre entradas pequeñas.

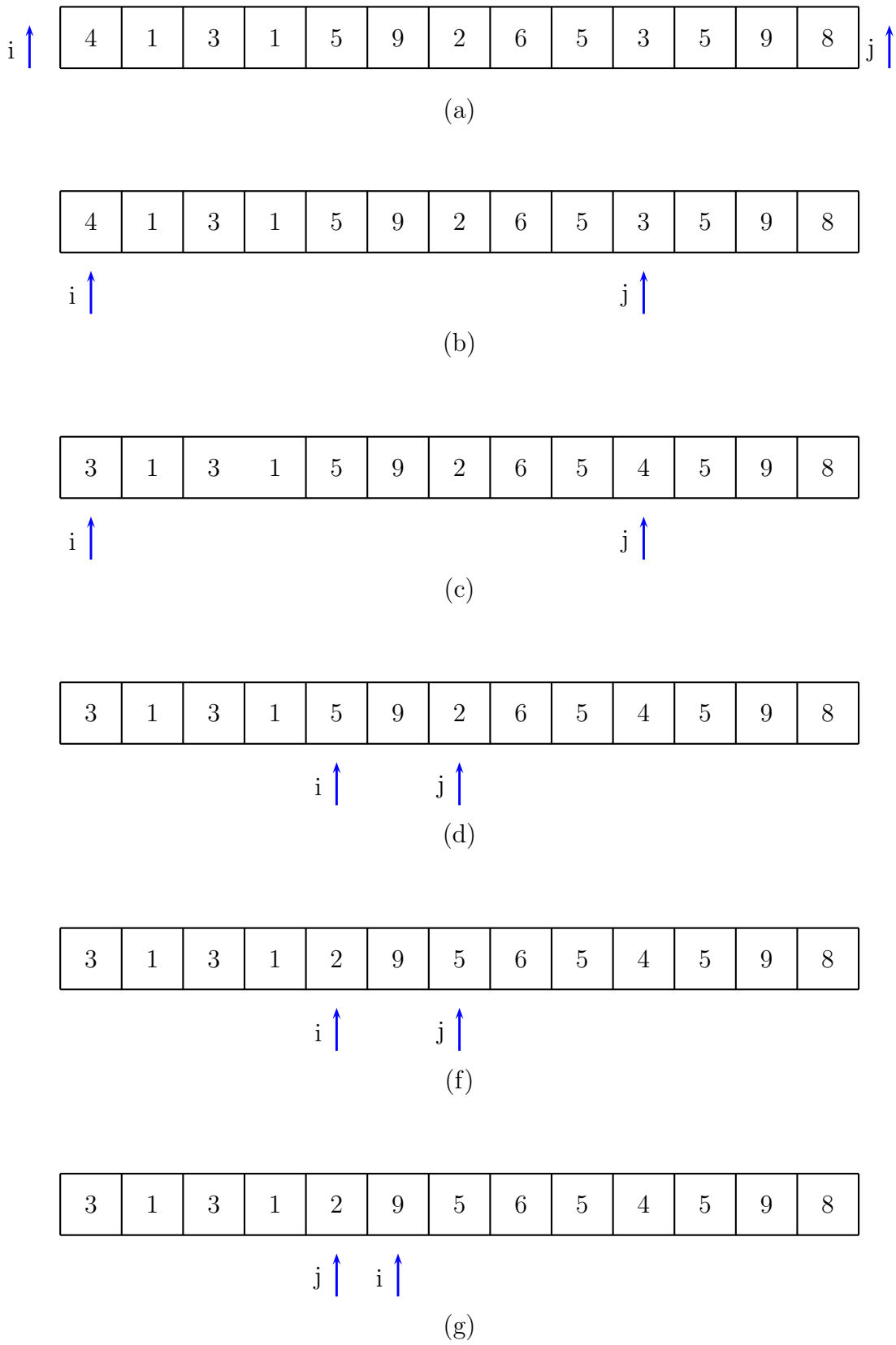


Figura 2.3: Procedimiento partición para el vector A del ejemplo 2.2.1

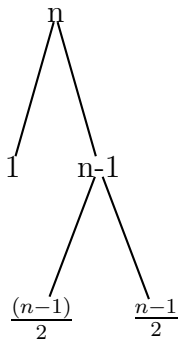


Figura 2.4: Dos niveles de el árbol de recursión para el procedimiento Partición

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$[19].

Para analizar el caso promedio de Quicksort se debe hacer una suposición de cómo se espera encontrar frecuentemente las diferentes entradas. Una suposición es que todas las entradas de números son igualmente probables. Cuando Quicksort corre sobre una entrada aleatoria es improbable que la partición se comporte de la misma manera en cada ejecución. Lo que se espera es que algunas divisiones sean razonablemente bien balanceadas y otras bastante desbalanceadas

En el caso promedio partición produce unas "malas y buenas"divisiones. Por ejemplo en la figura2.4 se muestran las divisiones en 2 niveles del árbol de recursión, para la raíz del árbol de recursión el costo es n para realizar la partición y los subvectores obtenidos son de tamaños 1 y $n-1$(el peor caso). Para el siguiente nivel el subvector de tamaño $n-1$ es particionado en dos subvectores de tamaños $\frac{(n-1)}{2}$(el mejor caso), así se obtienen tres subvectores de tamaños 1, $\frac{(n-1)}{n}$ y $\frac{(n-1)}{n}$ para un costo combinado de $2n-1 = \Theta(n)$. Intuitivamente se tiene que el costo $\Theta(n)$ de los buenos casos es absorbido por el costo $\Theta(n)$ de los malos casos y así las divisiones resultantes son buenas. De lo anterior se tiene que el tiempo de ejecución de Quicksort cuando los niveles alternan entre buenas y malas divisiones es semejante al tiempo de complejidad para las buenas divisiones, es decir, el tiempo de complejidad de Quicksort para el caso promedio es de $O(n \log n)$, pero con una

[19]Para solucionar este tipo de recurrencias hacemos uso de el método maestro (master) para recurrencias de la forma $T(n) = \alpha T(\frac{n}{b}) + f(n)$, [CLR90]

constante oculta grande por la notación O.

Ahora se verá la versión aleatoria de Quicksort que es la de interés en este documento, la cual tiene la propiedad de que el tiempo de complejidad del algoritmo es independiente de la entrada; para un caso partícular el peor caso del algoritmo no depende de la entrada, el peor caso depende de las opciones aleatorias que se tomen.

Modificando el procedimiento partición se puede diseñar una versión aleatoria de Quicksort[20] que usa la estrategia de las escogencias aleatorias; antes de particionar el vector intecambiamos el elemento $A[p]$ con un elemento escogido aleatoriamente de los $A[p..r]$, esta modificación significa que el pivote $x = A[p]$ es igualmente probable para cualquiera de los $r - p + 1$ elementos en el vector, así esperamos que las divisiones en el vector de entrada estén en promedio bien balanceadas.

Los cambios para partición y Quicksort son pequeños. En el nuevo procedimiento partición simplemente se utiliza la función random antes de particionar.

Partición-aleatorio(A, p, r)
inicio
$i \leftarrow Random(p, r)$
intercambie $A[p] \leftarrow A[i]$
devolver Partición(A, p, r)
fin.

Quicksort-aleatorio(A, p, r)
si $p < r$ **entonces**
inicio
$q \leftarrow$ Partición-aleatoria(A, p, r)
$Quicksort - aleatori(A, p, q)$

[20]Existe otra versión aleatoria de Quicksort que además de generar el pivote aleatoriamente, también toma las entradas aleatorias.

$Quicksort - aleatori(A, q+1, r)$

fin.

Ahora se analizará el caso promedio de este algoritmo. Para realizar el análisis el tiempo promedio esperado de Quicksort-aleatorio se debe entender primero como opera el procedimiento Partición-aleaotoria; se desarrollará una recurrencia para el tiempo esperado promedio requerido para ordenar un vector de n elementos y tal recurrencia será resuelta para obtener cotas sobre el tiempo esperado de ejecución.

Se deben tener en cuenta algunas observaciones acerca de la operación partición:

- El valor q devuelto por partición depende sólamente del rango[21] de $x = A[p]$ entre los elementos de $A[p..r]$.
- Si se considera $n = r - p + 1$ el número de elementos en $A[p..r]$, cambiando $A[p]$ con un elemento seleccionado aleatorio de $A[p..r]$. Se Tiene una probabilidad de $\frac{1}{n}$ que el $rango(x) = i$ para $i = 1, 2, \ldots, n$

Ahora, se calculan las probabilidades para las diferentes salidas de Partición.

- Si $rango(x) = 1$, la primera pasada por el ciclo mientras de Partición para el índice i se detiene en $i = p$ y el índice j se detiene cuando $j = p$. Así cuando $q = j$ es devuelto el lado inferior de la partición contiene sólo el elemento $A[p]$; este evento ocurre con probabilidad $\frac{1}{n}$, ya que esta es la probabilidad de que $rango(x) = 1$.
- Si $rango(x) \geq 2$ entonces hay al menos un elemento más pequeño que $x = A[p]$, luego en la primera pasada por el ciclo mientras de Partición, el índice i se detiene cuando $i = p$, pero j se detiene antes buscando a p. Hace un intercambio con $A[p]$ para colocarlo en la parte superior de la partición. Cuando Partición termina cada uno de los $rango(x) - 1$ elementos en la parte inferior son menores que x. Así para cada $i = 1, 2, \ldots, n-1$, cuando $rango(x) \geq 2$, la probabilidad es $\frac{1}{n}$ que al lado inferior de la partición tenga i elementos

[21] El rango de un número en un conjunto es el número de elementos menores o iguales a él.

combinando estos dos casos se concluye que para el tamaño $n = q - p + 1$ de un vector, el lado inferior de la partición es 1 con probabilidad $\frac{2}{n}$ y que el tamaño de la partición es i con probabilidad $\frac{1}{n}$ para $i = 2, 3, \ldots, n-1$.

Recurrencia para el caso promedio

Ahora se establece una recurrencia para el tiempo esperado de Quicksort-aleatorio. $T(n)$ denota el tiempo requerido para ordenar un vector de n elementos. Una llamada a Quicksort-aleatorio con un elemento en el vector de entrada toma un tiempo constante, así se tiene que $T(1) = \Theta(1)$. Una llamada a Quicksort-aleatorio con un vector $A[1..n]$ de longitud n usa un tiempo $\Theta(n)$ para particionar el vector, el procedimiento Partición devuelve un índice q y entonces Quicksort-aleatorio es llamado recursivamente con subvectores de longitud q y $n - q$, lo cual implica que el tiempo promedio para ordenar un vector de longitud n puede ser expresado como:

$$T(n) = \frac{1}{n}\Big(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\Big) + \Theta(n) \tag{2.5}$$

El valor de q tiene una distribución de probabilidad uniforme excepto cuando $q = 1$ que es dos veces probable. Si se obtienen dos subvectores de tamaños 1 y $n-1$ se estaría en el peor caso del algoritmo y ya que, por el ánalisis anterior se tiene que $T(n-1) = O(n^2)$ y $T(1) = \Theta(1)$, entonces
$\frac{1}{n}\Big(T(1) + T(n-1)\Big) = \frac{1}{n}\Big(\Theta(1) + O(n^2)\Big) = \frac{1}{n}O(n^2) = O(n)$

Además como $\Theta(n) = O(n) \bigcap \Omega(n)$, el termino $O(n)$ está en $\Theta(n)$, entonces se tiene que

$$\begin{aligned} T(n) &= \frac{1}{n}\Big(\sum_{q=1}^{n-1}(T(q) + T(n-q))\Big) + O(n) + \Theta(n) \\ &= \frac{1}{n}\Big(\sum_{q=1}^{n-1}(T(q) + T(n-q))\Big) + \Theta(n) \end{aligned}$$

Observe que para $k = 1, 2, \ldots, n-1$ cada término $T(k)$ en la sumatoria ocurre al menos una vez $T(q)$ y al menos una vez $T(n-q)$, uniendo los dos términos de la suma tenemos:

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} k + \Theta(n) \tag{2.6}$$

Para resolver (2.3) se asume que $T(n) \leq an \lg n + b$ para algunas constantes $a > 0$ y $b > 0$ por determinar. Se Puede asumir a y b suficientemente grandes tal que $an \lg n + b$ sea más grande que $T(1)$, porque el costo de $T(1)$ es constante, entonces para $n > 1$ se tiene:

$$\begin{aligned} T(n) &= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \leq \frac{2}{n}\Big(\sum_{k=1}^{n-1} ak \lg k + b\Big) + \Theta(n) \\ &= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n) \end{aligned}$$

Acotemos la sumatoria $\sum_{k=1}^{n-1} k \lg k$

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} k \lg k + \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} k \lg k$$

Como el logaritmo es una función creciente, en la primera sumatoria se tiene que:
$\lg k \leq \lg \frac{n}{2} = \lg n - 1$, lo cual implica que $\sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} k \lg k \leq (\lg n - 1) \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} k$, en la segunda sumatoria el logaritmo de $\lg k$ puede ser acotado por $\lg n$, entonces

$$\sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} k \lg k \leq \lg n \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} k$$

De lo anterior se tiene que,

$$\begin{aligned} \sum_{k=1}^{n-1} k \lg k &\leq (\lg n - 1) \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} k + \lg n \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} k = \\ &\lg n \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} k - \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} k + \lg n \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} k \\ &= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\frac{n}{2}} k = \lg n \Big(\frac{n(n-1)}{2}\Big) - \frac{n}{2}\Big(\frac{n}{2} - 1\Big)\frac{1}{2} = \\ &\frac{1}{2} \lg n \Big(n^2 - n\Big) - \Big(\frac{n^2}{4} - \frac{n}{2}\Big)\frac{1}{2} = \frac{1}{2} n^2 \lg n - \frac{1}{2} n \lg n - \frac{n^2}{8} + \frac{n}{4} \end{aligned}$$

Como $\frac{n}{4} < \frac{1}{2} n \lg n$ para $n \geq 2$ por que para $n \geq 2$ se tiene que:

$$
\begin{aligned}
2 < n^2 &= 2^{\lg(n^2)} \\
\lg 2 < \lg 2^{\lg(n^2)} &\Leftrightarrow 1 < 2\lg(n) \\
n < 2n\lg(n) &\Leftrightarrow \frac{n}{4} < \frac{1}{2}n\lg n \\
&\Leftrightarrow \frac{n}{4} - \frac{1}{2}n\lg n \ < \ 0
\end{aligned}
$$

Luego,

$$
\begin{aligned}
\sum_{k=1}^{n-1} k\lg k &\leq \frac{1}{2}n^2\lg n - \frac{1}{2}n\lg n - \frac{n^2}{8} + \frac{n}{4} \\
&\leq \frac{1}{2}n^2\lg n - \frac{n^2}{8}
\end{aligned}
$$

Así

$$
\begin{aligned}
T(n) \leq & \frac{2a}{n}\Big(\frac{1}{2}n^2\lg n - \frac{n^2}{8}\Big) + \frac{2b}{n}(n-1) + \Theta(n) \\
&= an\lg n - \frac{1}{4}an + 2b + \Theta(n) - \frac{2b}{n} \\
&\leq an\lg n - \frac{1}{4}an + 2b + \Theta(n) \\
&= an\lg n + b + \Big(\Theta(n) + b - \frac{an}{4}\Big)
\end{aligned}
$$

Puesto que se puede escoger a suficientemente grande tal que $\frac{an}{4}$ domine a $\Theta(n)$. Se Puede concluir que el tiempo esperado promedio del algoritmo Quicksort-aleatorio está en $O(n\lg n)$.

Así, se tiene un algoritmo probabilístico para ordenar n elementos de un vector en forma no decreciente, cuyo tiempo esperado promedio es del orden de $O(n\lg n)$, el cual depende sólamente del la longitud del vector, pero no de la entrada en particular.

2.2.2. Factorización de enteros grandes

Del teorema fundamental de la aritmética se sabe que cada entero positivo n se puede escribir como producto único de números primos. El problema de encontrar la factorización

de un entero grande es tan antiguo como el problema de decidir si un número entero es o no primo; aún no se conocen algoritmos probabilísticos (con probabilidad mayor que $\frac{1}{2}$ de encontrar un factor en tiempo polinomial) ni determinísticos que sean eficientes (polinomiales) para resolver este problema. En esta sección se discutirá el problema de encontrar esta factorización.

Antes de abordar el problema de la factorización se darán algunos resultados:

Teorema 2.2.1. *[KEN97] Si a,b y c son enteros con $a|b$ y $b|c$, entonces $a|c$*

Teorema 2.2.2. *[KEN97] Si n es un número entero compuesto, entonces n tiene un factor primo que no excede a $\sqrt{n}$*

Demostración. Como n es compuesto existen enteros a y b con $1 < a \leq b < n$ tales que $n = ab$. Se debe tener que $a \leq \sqrt{n}$, porque de lo contrario.

$$\sqrt{n} < a \leq b, \text{ implica } \sqrt{n}a < ab \text{ luego, } \sqrt{n}\sqrt{n} < ab, \text{ de ahí que } n < ab$$

lo cual es un contradicción.
Ahora como $a > 1$, a debe tener un divisor primo, sea p este divisor, luego como $p|a$ y $a|n$ por el teorema anterior $p|n$. Además $p \leq a \leq \sqrt{n}$, entonces, $p \leq \sqrt{n}$, es decir, n tiene un factor primo que no excede a $\sqrt{n}$. $\diamondsuit$

La forma más directa de encontrar la factorización de un entero positivo n, si n no es primo, es encontrar un divisor no trivial m de n y factorizar recursivamente m y $\frac{n}{m}$. Por el teorema (2.2.2) es útil buscar divisores de n menores que $\sqrt{n}$, lo anterior lleva a pensar en una solución ingenua para el problema de factorización, la cual será mostrada en los siguientes algoritmos: el primero halla el menor primo que es divisor de n, y el segundo busca recursivamente los divisores de $\frac{n}{m}$, para así hallar la factorización completa de n. Este procedimiento se conoce como el algoritmo de factorización basado en la prueba de la división.
Función divisores(Entrada: n)
Inicio
$m \leftarrow 2$

Mientras $m < \lceil \sqrt{n} \rceil$ **hacer**

si m divide a n **entonces**

devolver m

$m \leftarrow m + 1$

devolver n

fin.

procedimiento factorizar(n, **var** fact)

Inicio

$i \leftarrow 1$

$p \leftarrow n$

mientras $divisores(p) \neq p$ **hacer**

inicio

$fact[i] \leftarrow divisores(p)$

$p \leftarrow \frac{p}{divisores(p)}$

$i \leftarrow i + 1$

fin

$fact[i] \leftarrow p$

fin.

El algoritmo anterior necesita para factorizar un entero n a lo más $\sqrt{n}$ divisiones[22]. La expresión $\sqrt{n}$ representada en binario utilizaría 2^k bits, donde $k = \lceil \lg \sqrt{n} - 1 \rceil$, lo cual es una función exponencial que depende del número de dígitos de n.

De lo anterior se tiene que el algoritmo de factorización basado en la prueba de la división es ineficiente. Por ejemplo para un número "duro"[23] de 40 dígitos el algoritmo puede tardar miles de años, si cada ejecución del ciclo mientras tarda un nanosegundo (10^{-9} segundos).

[22]Recuerde que para representar un número n en binario son necesarios 2^k bits donde $k = \lceil \lg n - 1 \rceil$.

[23]Número "duro"significa que es el producto de dos primos de tamaños casi iguales.

Numeros de dígitos	tiempo empleado
50	3.9 horas
75	104 dás
100	74 años
200	3.8×10^{9} años
300	4.9×10^{15} años
500	42×10^{25} años

tabla 2.4: Tiempo empleado para la factorización de enteros de diferente número de dígitos

Es por esto que el problema de calcular factores primos de un entero es muy costoso computacionalmente. Esa dificultad es la razón por la que se siguen utilizando actualmente los sistemas de criptografia[24] de clave pública, como por ejemplo RSA, cuya inviolabilidad descansa en el hecho de que no se conoce un método que permita factorizar eficientemente (en un tiempo razonable y con medios no demasiado costosos) un entero que sea el producto de primos grandes.

Existen varios algoritmos para este problema: Factorización usando curvas elípticas, ρ de Pollar, de Fermat, Euler, entre otros, los cuales son complicados y no totalmente eficientes. Por ejemplo, en la tabla 2.4 se muestra el tiempo necesario para factorizar enteros de varios tamaños, utilizando el algoritmo más eficiente conocido en 1986 [KEN97]; como se puede ver en la tabla el problema de factorizar un entero n es muy costoso.
A continuación se dará la idea de un algoritmo de descomposición "eficiente". El algoritmo esta basado en el siguiente teorema

Teorema 2.2.3. *Sea n un entero compuesto. Sean a y b dos enteros distintos entre 1 y $n-1$ tal que $a+b \neq n$. Si $a^2 \cong b^2 \pmod{n}$, entonces $mcd(a+b,n)$ es un divisor no trivial de n.*

Demostración. Sean a y b enteros distintos entre 1 y $n-1$, por hipótesis se tiene que $a^2 \cong b^2 \pmod{n}$, luego

[24]Es el arte y la ciencia de la comunicación secreta a través de canales poco seguros.

$$a^2 \cong b^2 \pmod{n} \Leftrightarrow a^2 - b^2 \cong 0 \pmod{n}$$
$$n|a^2 - b^2 \Leftrightarrow n|(a-b)(a+b)$$

De lo anterior, $n|a+b$ *o* $n|a-b$
Si $n|a-b$, entonces $a - b \cong 0 \pmod{n}$, lo cual equivale a que $a \cong b \pmod{n}$, como a y b son enteros entre 1 y $n-1$, la ecuación $a - b \cong 0 \pmod{n}$ implica que $a = b$, lo cual es una contradicción.

Luego, sólo se da que $n|a+b$, por el teorema fundamental de la aritmética todo entero se puede escribir como producto único de primos. Entonces n se puede escribir de la forma $n = p_1^{k_1} p_2^{k_2} \dots p_r^{k_r}$, donde $k_i \geq 0$ para toda $i = 1, 2 \dots, r$ entonces

$$n|a+b \Leftrightarrow p_1^{k_1} p_2^{k_2} \dots p_r^{k_r}|a+b$$

Entonces, existe $t \in \mathbb{Z}$ tal que $(p_1^{k_1} p_2^{k_2} \dots p_r^{k_r})t = a+b$, equivalentemente, $p_i^{k_i}(t \prod_{i=1, i\neq j}^{r} p_i^{k_i}) = a+b$, sea $t_j = t \prod_{i=1, i\neq j}^{r} p_i^{k_i}$, claramente $t_j \in \mathbb{Z}$ por ser producto de números enteros.

Entonces, como $p_i^{k_i} t_j = a+b$ se tiene que $p_i^{k_i}|a+b$, por un análisis semejante al anterior se tiene que $p_i|a+b$.

Además como $a+b \neq n$ y por ser compuesto n este se puede expresar al menos como potencia de un primo y además se tiene que: $p_i|a+b$ y $P_i|n$.
Entonces, $1 < mcd(a+b, n) < n$, esto es el $mcd(a+b, n)$ es un divisor no trivial de n. $\diamondsuit$

Del teorema se tiene una idea para descomponer n. Se debe buscar dos enteros entre 1 y $n-1$ distintos que tengan el mismo cuadrado módulo n, pero cuya suma no sea n, y utilizar el algoritmo de Euclides para calcular el mcd de su suma y n. La idea del teorema es buena siempre que tales números existan y puedan ser buscados eficientemente.

procedimiento *Factorización-Vegas*(Entrada: n, k **var** $fatV$)

inicio

$i \leftarrow 1$

repetir

congruencia(n, k) {devuelve dos enteros a y b
entre 1 y $n-1$ tal que tengan el mismo cuadrado módulo n}

hasta $a \neq b$ y $a + b \neq n$

$y \leftarrow Euclides(a+b, n)$

$factV[i] \leftarrow y$

factorizar($\frac{n}{y}, fact$)

para $j \leftarrow 1$ **hasta** longitud (fact) **hacer**

$factV[j+1] \leftarrow fact[j]$

fin.

La función importante en el algoritmo anterior es *congruencia*; para garantizar que tales números a y b existen enunciamos un resultado importante de teoría de números, el teorema chino del resto.

Teorema chino del resto. *[JGR99] Sean $m_1, m_2, \ldots, m_r$ enteros positivos primos relativos dos a dos. Entonces el sistema de congruencias lineales*

$$\begin{aligned} x &\cong a_1 (mod\, m_1) \\ x &\cong a_2 (mod\, m_2) \\ &\vdots \\ x &\cong a_r (mod\, m_r) \end{aligned}$$

Tiene solución unica módulo $m = \prod_{i=1}^{r} m_i$ [25]

La existencia de dos números entre 1 y $n-1$ tales que sus cuadrados sean congruentes módulo n está garantizada porque si $x = b^2$ y $mcd(a, n) = 1$ entonces la ecuación $a^2 \cong x$ (mód n) admite al menos 4 soluciones distintas en aritmética módulo n. En efecto, sea $n = pq$ donde p y q son primos impares distintos, entonces la congruencia:
$a^2 \cong x \pmod{p}$ tiene dos soluciones $a \cong b \pmod{p}$ y $a \cong p - b \pmod{p}$.

[25] Este resultado y su demostración pueden ser vistos en: [KEN97, JGR99]

De manera similar la congruencia $a^2 \cong x \pmod{q}$ tiene dos soluciones incongruentes módulo q $a \cong y \pmod{q}$ y $a \cong q - y \pmod{q}$. Por el teorema chino del resto hay exactamente 4 soluciones incongruentes módulo n de la ecuación $a^2 \cong x \pmod{n}$. Estas cuatro soluciones incongruentes módulo n son las soluciones de los sistemas.

$$\begin{aligned} a \cong b(mod\, p) \text{y} a \cong y(mod\, q) \quad a \cong p - b(mod\, p) \text{y} a \cong y(mod\, q) \\ a \cong b(mod\, p) \text{y} a \cong q - y(mod\, q) \quad a \cong p - b(mod\, p) \text{y} a \cong q - y(mod\, q) \end{aligned} \tag{2.7}$$

¿Y entonces, como se puede hallar a y b con la propiedad deseada?, éste es el momento en que entra en juego la aleatoriedad. Primero se dará una definicición que será de mucha utilidad.

Definición 2.8. *Sea k un entero, se dice que un número n es $k - uniforme$ si todos sus divisores primos se encuentran entre los primeros k números primos más pequeños.*

El algoritmo *congruencia* selecciona aleatoriamente un entero x entre 1 y $n - 1$, y calcula $y = x^2(mod\, n)$, es decir se calcula el cuadrado módulo n del número seleccionado aleatoriamente. Si y es k-uniforme entonces x y la factorización de y se almacenan en una tabla. En caso contrario se selecciona aleatoriamente otro número x; el proceso se repite hasta que se hayan encontrado $k + 1$ enteros diferentes para los cuales se conoce la factorización de sus cuadrados módulo n.

función *congruencia*(Entrada: n, k)
inicio
$cont \leftarrow 0$
repetir
$x \leftarrow Random(1, n - 1)$
$y \leftarrow x^2 \pmod{n}$
factorizar(y, fact)
si uniforme(k, fact(y))=verdadero **entonces**
inicio
adicionar x, y, y $fact(y)$

$cont \leftarrow cont + 1$

fin

hasta $cont \leq k + 1$

Con las y y los primos k-uniformes, formar una matriz de $\{0, 1\}$. Encontrar un conjunto no vacío de filas que sumen cero módulo 2, multiplicar las y_i correspondientes a cada fila seleccionada, sacar raíz cuadrada de este producto llamar a el resultado obtenido. Multilplicar las x_i correspondientes a las y_i seleccionadas para así obtener otra raíz cuadrada del mismo número, llamar b este resultado.

devolver a y b

fin.

Este algoritmo encuentra a y b tal que $a^2 \cong b^2 \pmod{n}$, se probará que la probabilidad de que $a+b = n$ es menor que $\frac{1}{2}$. Suponga que $n = p_1^{k_1} p_2^{k_2} \ldots p_n^{k_r}$, $k_i \geq 0$, entonces se forman 2^r sistemas de ecuaciones, como el sistema mostrado en las ecuaciones 2.7. Entonces la probabilidad que $a + b = n$ esta dada por.

$$prob[a + b = n] = \frac{2}{2^r} = 2^{1-r} < \frac{1}{2}$$

luego, la $prob[a + b \neq n] > 1 - \frac{1}{2} = \frac{1}{2}$.

El algoritmo para verificar si un entero dado es k- uniforme es dado a continuación.

función *uniforme*(Entrada: k, factores)

inicio

$j \leftarrow 1$

$i \leftarrow 1$

$cont \leftarrow 0$

primos $\leftarrow$ los primeros k números primos

mientras $i \leq longitud(factores)$ **hacer**

repetir

$j \leftarrow j + 1$

hasta $j = k + 1$ o $primos[j] = factores(i)$

si $primos[j] = factores(i)$ **entonces**

inicio

$cont \leftarrow cont + 1$

$i \leftarrow i + 1$

fin

si $j = k + 1$ **entonces**

devolver "falso"

si $cont = longitud(factores)$ **entonces**

devolver "verdadero"

fin

El algoritmo de las Vegas para el problema de la factorización de enteros es un ejemplo de los algoritmos de las Vegas del segundo tipo, es decir de aquellos que si no encuentran la solución vuelven a reiniciar el algoritmo sobre la misma entrada para tener una nueva posibilidad de éxito, sin embargo para este algoritmo se da una mejor alternativa que reiniciar todo, ya que no hay necesidad de desperdiciar tanto trabajo bien hecho.

Para aclarar el algoritmo *Factorizar-Vegas* se dará el siguiente ejemplo:

Ejemplo 7.

El algoritmo *factorización-vegas* es llamado con n =1729 y $k = 7$. Este algoritmo inicialmente llama a la función *congruencia* con los parámetros n y k. Esta función selecciona aleatoriamente un entero entre 1 y 1728. Si el algoritmo selecciona aleatoriamente el entero $x = 1711$ y calcula su cuadrado módulo n. Se tiene que $x^2 \pmod{n} = (1711)^2 \pmod{1}729 = 324$, factorizando el número 324 utilizando la prueba división mostrada en la primera parte de esta sección, se tiene que $324 = 2^2 \times 81$. En este momento el algoritmo llama la función *uniforme* la cual esta considerando los primos $\{2, 3, 5, 7, 11, 13, 17\}$. Pero, uniforme(7, $2^2 \times 81$)= falso, en este caso no se tiene éxito porque 324 no es 7-uniforme.

Un segundo intento con $x = 1623$ es más afortunado su cuadrado módulo n es $30 = 2\times3\times5$ el cual es 7-uniforme; prosiguiendo así hasta conseguir 8 éxitos se tiene la siguiente tabla.

$$
\begin{array}{lllllll}
x_1 & = & 1623 & y_1 & = & 30 & = & 2 \times 3 \times 5 \\
x_2 & = & 1105 & y_2 & = & 351 & = & 3^3 \times 13 \\
x_3 & = & 1727 & y_3 & = & 4 & = & 2^2 \\
x_4 & = & 1321 & y_4 & = & 480 & = & 2^5 \times 3 \times 5 \\
x_5 & = & 225 & y_5 & = & 484 & = & 2^2 \times 11^2 \\
x_6 & = & 1705 & y_6 & = & 144 & = & 2^4 \times 3^2 \\
x_7 & = & 1357 & y_7 & = & 64 & = & 2^6 \\
x_8 & = & 1203 & y_8 & = & 36 & = & 2^2 \times 3^2
\end{array}
$$

La tabla se utiliza para formar una matriz M de $(k+1) \times k$ sobre $\{0, 1\}$; cada fila corresponde a un éxito y cada columna corresponde a un primo admisible. La entrada m_{ij} es cero si el j-ésimo primo aparece como potencia par (incluyendo 0) en la factorización de y_i, en caso contrario $M_{ij} = 1$. Por ejemplo $M_{34} = 1$ porque el cuarto primo, que es 7, aparece en la factorización de y_3 como potencia impar.

Luego la matriz A esta dada por:

	2	3	5	7	11	13	17
y_1	1	1	1	0	0	0	0
y_2	0	1	0	0	0	1	0
y_3	0	0	0	0	0	0	0
y_4	1	1	1	0	0	0	0
y_5	0	0	0	0	0	0	0
y_6	0	0	0	0	0	0	0
y_7	0	0	0	0	0	0	0
y_8	0	0	0	0	0	0	0

Se debe encontrar un conjunto no vacío de filas cuya suma valga cero en aritmética módulo 2, tome el conjunto formado por las filas $\{y_1, y_4\}$. Multiplicando las y_i correspondientes al conjunto de filas seleccionadas se obtiene:

$$y_1 y_4 = 2 \times 3 \times 5 \times 2^5 \times 3 \times 5 = 2^6 \times 3^2 \times 5^2$$

Los exponentes de estos números se han construido de tal manera que sean pares. Luego se puede obtener una raíz de estos productos dividiendo los exponentes por dos; También

se puede obtener otra raíz cuadrada del producto las x_i correspondientes a las filas seleccionadas, porque por construcción $y_i = x_i^2 \pmod n$. Los números obtenidos a y b son de la forma:

$$a = 2^3 \times 3 \times 5 \pmod{1729} = 120$$

$$b = 1623 \times 1321 \pmod{1729} = 23$$

Se han encontrado dos enteros $a = 120$ y $b = 23$ entre 1 y $n - 1$ tal que $a^2 \cong b^2$ (mód 1729), luego el algoritmo congruencia devuelve:

$$a = 120 \quad \text{y} \quad b = 23$$

Como $a \neq b$ y $a + b \neq n$, continuando con el algoritmo *Factorización-Vegas* se llama *euclides(1729, 143)* y el algoritmo devuelve $y = 3$. En este momento el algoritmo factoriza

$$\frac{n}{y} = \frac{1729}{13} = 133$$

Por lo tanto, el algoritmo *factorización-Vegas* devuelve $facV = [13, 17, 19]$, es decir $1729 = 13 \times 17 \times 19$.

Para determinar el valor de k que se debe utilizar para optimizar el rendimiento de esta aproximación se debe tener en cuenta que: cuanto mayor sea k mayor será la probabilidad de que $x^2 (mod\, n)$ sea k-uniforme cuando seleccionemos x aleatoriamente. De otra manera cuanto más pequeño sea el parámetro k más rápido realizaremos la prueba de k-uniformidad y también la factorización de los valores encontrados se hará más rápidamente. la búsqueda de un k que optimice la aproximación de las Vegas requiere el uso de resultados avanzados de teoría de números.[BB97]
Sea

$$L = e^{\sqrt{\lg\, n \lg\lg\, n}}$$

Si se toma $k \approx \sqrt{L}$ el algoritmo factoriza un número n al cabo de un número esperado de pasos que está en $O(L^2)$. Si n es un número de 100 dígitos decimales, entonces $L^2 \approx 5 \times 10^{30}$ mientras que $\sqrt{n} \approx 7 \times 10^{49}$ que es 10^{19} veces mayor. De lo anterior se puede ver que la aproximación de las Vegas es mucho mejor que el algoritmo de factorización

basado en la prueba de la división, pero incluso 10^{30} picosegundos son aproximadamente el doble de la edad estimada del universo.[BB97]

La aleatoriedad desempeña un papel fundamental en este algoritmo porque, no hay ninguna aproximación determinista que haya resultado ser eficiente, para hallar tantos x buenos.

3. MÁQUINAS DE TURING PROBABILÍSTICAS Y SUS CLASES DE COMPLEJIDAD

Intuitivamente, el significado de computar es, ser capaz de especificar una secuencia finita de pasos con el fín de obtener algún resultado. El intento de muchos matemáticos de formalizar la noción de algoritmo, los llevo a desarrollar muchos modelos distintos que finalmente fueron mostrados equivalentes, en poder computacional, a la máquina de Turing.

La teoría de la computabilidad desarrollada principalmente por Alan Turing, Alonso Church y Kurt Gödel a comienzos de la década de los $30's$, fué la que dió origen a la teoría de la complejidad computacional. Dicha teoría se preocupaba por conocer si toda función es computable[26], lo cual llevó a Turing a desarrollar una máquina llamada **máquina de Turing**, capaz de computar funciones. Turing no se preocupó por la eficiencia de sus máquinas, esto es, por el tiempo usado en sus computaciones, sólo se interesó en que su máquina fuese capaz de simular algoritmos arbitrarios.

La definición clásica de algoritmo determinístico es:

Definición 3.1. *Un algoritmo determinístico es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.*

Cuando se utiliza un algoritmo determinístico para encontrar la respuesta de un problema concreto, debe suceder que al realizar la serie de pasos, si se aplican correctamente,

[26]Una función es computable se existe un procedimiento que, siguiendo un conjunto finito de pasos, encuentra el valor de la función para cualquier entrada.

se obtendrá la respuesta correcta. Un algoritmo determinístico debe ser preciso, esto es, debe indicar el orden de realización de cada paso; además, los pasos que se siguen deben ser definidos en el sentido que si el algoritmo se sigue dos veces (con la misma entrada), se obtiene la misma respuesta cada vez. Por último un algoritmo debe tener un número determinado de pasos y producir un resultado en un tiempo finito.

Existe otra clase de algoritmos que no cumplen con todas las características de la definición anterior, los **algoritmos probabilísticos**, se diferencia de los algoritmos descritos arriba, fundamentalmente, en que un mismo algoritmo probabilístico se puede comportar en forma distinta cuando se aplica dos veces a una misma entrada. Además, alguno de sus pasos es aleatorio. A continuación se da la definición de esta clase de algoritmos.

Definición 3.2. *Un algoritmo probabilístico es un procedimiento para resolver un problema mediante una serie de pasos precisos y finitos, de los cuales alguno es aleatorio.*

De la definición se vé claramente que los algoritmos probabilísticos pueden dar respuestas equivocadas, aunque esto debe suceder con probabilidad suficientemente pequeña, como se ha mostrado en los capítulos 1 y 2.

Para formalizar el concepto de algoritmo convencional, se considera la máquina de Turing deteminística (MTD). También se estudiarán las máquinas de Turing no determinísticas (MTN)[27], para así llegar a definir el modelo de computación probabilística, máquinas de Turing probabilísticas, las cuales serán usadas para formalizar los algoritmos probabilísticos y definir las diferentes clases de complejidad probabilísticas.

3.1. MÁQUINAS DE TURING PROBABILÍSTICAS

Para iniciar el estudio del modelo probabilístico de computación, máquina de Turing probabilística (MTP), se hace necesario conocer los modelos computacionales, máquinas de Turing determinísticas (MTD) y máquinas de Turing no determinísticas (MTN); primero se darán algunas definiciones:

[27]Este tipo de máquinas se estudian detalladamente en [HB03]

Definición 3.3. *Un alfabeto Σ es un conjunto no vacío y numerable (finito o infinito) de símbolos indivisibles.*

Un ejemplo muy conocido de alfabeto es el utilizado por la computadora actual, compuesto por dos elementos que son el cero y el uno, es decir $\Sigma = \{0, 1\}$, los cuales son usados por la máquina para el manejo de la información.

Definición 3.4. *Una cadena es una sucesión finita de símbolos de un alfabeto Σ. Como cadena de Σ se incluye la cadena vacía denotada por ε*

Definición 3.5. *El conjunto Σ^* es el conjunto de todas las cadenas que se pueden construir sobre un alfabeto Σ, esto es,*

$$\Sigma^* = \{a_1 a_2 \dots a_n : n \in \mathbb{Z}\} \cup \{\varepsilon\}, \qquad a_i \in \Sigma$$

Definición 3.6. *Sea Σ un alfabeto. Un lenguaje L sobre Σ denotado $L(\Sigma)$ es un subconjunto de Σ^*, es decir $L(\Sigma)$ es un lenguaje sobre Σ, si y sólo si, $L(\Sigma) \subseteq \Sigma^*$*

Una máquina de Turing determinística es un modelo computacional abstracto que captura y formaliza la noción de algoritmo. Estas máquinas pueden ser usadas para computar funciones, resolver problemas de desición, entre otras.

La entrada de una máquina de Turing es una cadena finita de símbolos de un determinado alfabeto, el cual por convención siempre contendrá el símbolo especial llamado "símbolo blanco"y denotado por b.

La máquina utiliza para la entrada y salida de datos, una cinta bi-infinita dividida en celdas etiquetadas por $\dots, -2, -1, 0, 1, 2, \dots$, las cuales sólo pueden contener un símbolo por celda.

La máquina tiene una cabeza de lectura-escritura que se puede mover libremente, una celda en cada paso y tiene la capacidad de leer o escribir un símbolo en la celda. La máquina también posee un conjunto finito de estados.

Una *situación* de la máquina se define como una pareja formada por un estado y un símbolo del alfabeto. En un instante de tiempo, la situación actual de la máquna esta determinada por el estado actual en que se encuentra y por el símbolo que está leyendo actualmente la cabeza de lectura-escritura.

La parte fundamental de la máquina de Turing es el conjunto de instrucciones que representan el comportamiento de la máquina en cada instante de tiempo. Cada instrucción está formada por dos partes: la primera por un estado y un símbolo e indica en qué situación se debe ejecutar la instrucción. La segunda parte, indica lo que hace la máquina en esa situación, lo cual consiste en colocar la máquina en un nuevo estado, escribir un símbolo en la posición actual y mover la cabeza de lectura-escritura.

Con lo anterior, se puede definir las máquinas de Turing determinísticas y las no determinísticas.

Definición 3.7. *Una máquina de Turing determinística (MTD) es una terna* (Q, Σ, δ) *donde:*

1. Q *es un conjunto finito, llamado el conjunto de estados, el cual contiene un estado especial* q_0*, llamado el estado inicial y tres estados de parada: el estado de aceptación* q_Y*, el estado de rechazo* q_N*, y el estado stop.*

2. Σ *se define como el conjunto finito de símbolos llamado el alfabeto de la máquina, el cual siempre contiene el símbolo blanco b.*

3. δ *es una función llamada función de transición definida de un subconjunto de* $(Q - \{q_y, q_N, stop\}) \times \Sigma$ *en* $Q \times \Sigma \times \mathtt{D}$*, donde* $\mathtt{D} = \{I, D, N\}$ *es el conjunto de direcciones (I: izquierda, D: derecha, N: ninguno). La función* δ *puede ser representada como un conjunto finito de instrucciones, donde cada instrucción es una quíntupla de la forma* (q, s, q', s', d)*, donde* $q, q' \in Q$*,* $s, s' \in \Sigma$ *y* $d \in \mathtt{D}$ *tal que ningún par de quíntuplas en el conjunto tienen los dos primeros elementos iguales.*

Si la situación actual de la máquina en algún instante de tiempo es (q, s) y $\delta(q, s) = (q', s', d)$, entonces la máquina pasa del estado q al estado q', la cabeza de lectura-escritura

q	s	$\delta(q,s)$
q_0	0	$(q_0, 0, d)$
q_0	1	$(q_0, 1, d)$
q_0	b	$(q_1, 0, d)$
q_1	b	$(stop, 0, N)$

tabla 3.5: Función de transición de la MTD

borra s y escribe s' en su lugar y se mueve a la izquierda si $d = I$, a la derecha si $d = D$, o no se mueve si $d = N$. Esto constituye un paso en la ejecución de la máquina y produce una nueva situación actual (q', s') con la cual se procede de igual manera, aplicando la función de transición.

La ejecución de la máquina termina si alcanza uno de los tres estados de parada q_Y, q_N o *stop*. Si el estado alcanzado es q_N se dice que la máquina para, rechazando la entrada. Si el estado alcanzado es q_Y, la máquina para, aceptando la entrada. Si lo que se quiere es conocer la cadena obtenida al final de la ejecución de la máquina; por ejemplo, cuando se quiere computar una función, se utiliza el estado *stop*. Es decir, si la máquina de Turing alcanza el estado *stop*, la máquina para y la salida es la cadena contenida en la cinta al momento de la parada.

Ejemplo 8.
Considere la máquina de Turing $MTD = (Q, \Sigma, \delta)$, donde $Q = \{q_0, q_1, q_Y, q_N, stop\}$, $\Sigma = \{0, 1, b\}$ y δ es la función de transición representada en la tabla 3.5. Esta máquina toma una entrada $x = 111$, escrita en binario y la multiplica por 4. Las computaciones de la máquina se representan en la figura 3.5.

Definición 3.8. *Una máquina de Turing no determinística es una terna* (Q, Σ, δ) *donde:* Q *y* Σ *son los mismos conjuntos para una máquina determinística y* δ *es una relación*

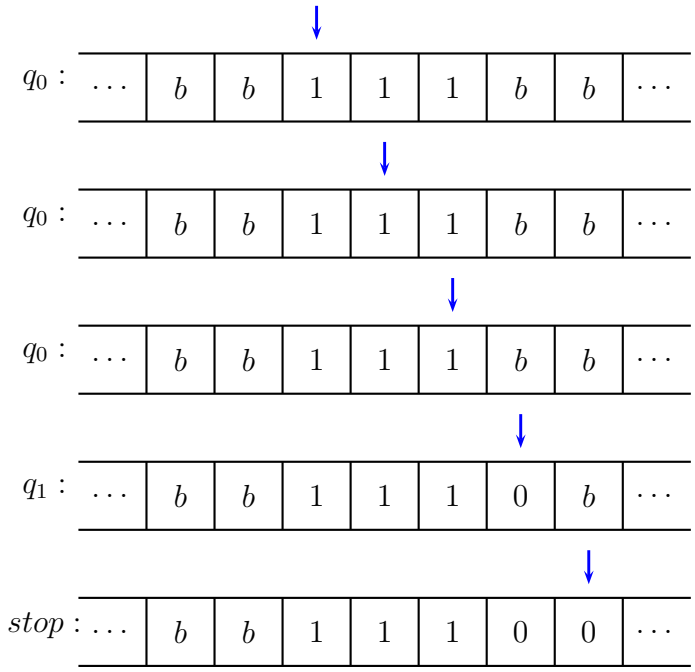


Figura 3.5: Una computación de MTD dada por la función de transición representada en la tabla 3.5, para la entrada $x = 111$

definida de $(Q - \{q_Y, q_N, stop\}) \times \Sigma$ *en* $Q \times \Sigma \times D$*, es decir*

$$\delta \subseteq (Q - \{q_Y, q_N, stop\}) \times \Sigma \times Q \times \Sigma \times \mathtt{D}$$

δ refleja la principal diferencia entre una máquina de Turing deterministica y una máquina de Turing no deterministica; en la primera δ es un función mientras que en la segunda δ es una relación. Esto significa que para una situación particular de la máquina de Turing no deterministica, no éxiste sólo instrucción a seguir sino que la MTN puede escoger entre varias posibilidades, la instrucción a ejecutar.

Una computación no deterministica sobre una entrada x, $MTN(x)$, puede ser vista como un árbol llamado *árbol de computación*; los nodos corresponden a las situaciones de la máquina y las aristas entre los nodos corresponden a una aplicación de δ. Cada camino finito o infinito, comenzando desde la raíz, se denomina *camino de computación.* En la figura 3.6 se muestran los tres primeros niveles para una MTN aplicada a una entrada x

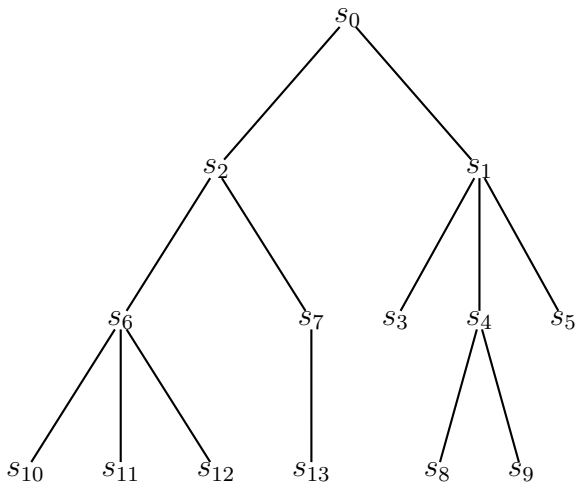


Figura 3.6: Primeros tres niveles del árbol de computación asociado una $MTN(x)$

Ahora, se define el grado del árbol de computación de una MTN y el lenguaje aceptado por dicha máquina.

Definición 3.9. *Sea MTN una máquina de Turing no determinística. Para cada $(q, s) \in Q - \{q_Y, q_N, stop\} \times \Sigma$, sea $C_{q,s} = \{(q', s', d') : (q, s, q', s', d') \in \delta\}$. Se define el grado del árbol de computación de la MTN como:*

$$d = max|C_{q,s}| \qquad para \qquad (q, s) \in (Q - \{q_Y, q_N, stop\})$$

Definición 3.10. *Sea MTN una máquina de Turing no determinística. Una entrada x es aceptada por MTN, si el árbol de computación asociado con $MTN(x)$ incluye por lo menos un camino de computación de aceptación, esto es, un camino de computación finito cuya hoja es una situación que contiene el estado q_Y. El conjunto de entradas aceptadas por MTN es llamado el lenguaje aceptado por MTN, y se denota $L(MTN)$.*

Cada uno de los modelos computacionales definidos anteriormente se utiliza para definir las clases se complejidades P y NP, respectivamente, como sigue:

Definición 3.11. *La clase P es el conjunto de todos los lenguajes que son aceptados por una máquina de Turing determinística tiempo polinomial, esto es:*

$$P = \big\{ L : L \text{ es un lenguaje y existe una máquina de Turing deterministica } M \text{ que corre en tiempo polinomial y tal que } L = L(M) \big\}$$

La clase NP[28] se define a continuación.

Definición 3.12. *La clase de complejidad NP se define como el siguiente conjunto:*

$$NP = \{L : L \text{ es un lenguaje y existe una máquina de Turing no deterministica } M \text{ que corre en tiempo polinomial y tal que } L = L(M)\}$$

Con todo lo anterior se hace fácil la comprensión de el modelo probabilístico de computación ya que éste es visto como una máquina de Turing no determinística tiempo polinomial, en la cual se deben tener en cuenta algunas condiciones y con un criterio de aceptación diferente, unido a los ya conocidos $q_Y, q_N, stop$. La definición de este tipo de máquinas de Turing se da a continuación.

Definición 3.13. *Una máquina de Turing probabilística (MTP) se define como una máquina de Turing no determinística en la cual se tiene que:*

1. *El grado de la máquina es 2(la relación δ sólo puede escoger entre dos opciones, la situación a ejecutar).*
2. *Cada paso de la máquina es aleatorio.*
3. *Todas las computaciones sobre una entrada x paran, después del mismo número de pasos.*
4. *El criterio de aceptación es diferente.*
5. *Tiene un nuevo estado de parada, adicionado a los ya conocidos de aceptación y rechazo, el estado "no sé".*

De la definición anterior se sigue que para cualquier entrada x, el árbol de computación asociado a $MTP(x)$ es un árbol binario perfecto, ya que la relación δ sólo puede escoger entre dos posibilidades la instrucción a ejecutar y además todas las computaciones paran después del mismo número de pasos, esto es, en el árbol de computación asociado con la entrada x ningún camino de computación es más largo.

[28] La definición de la clase NP también se puede dar en términos de algoritmos de verificación, ver en [HB03]

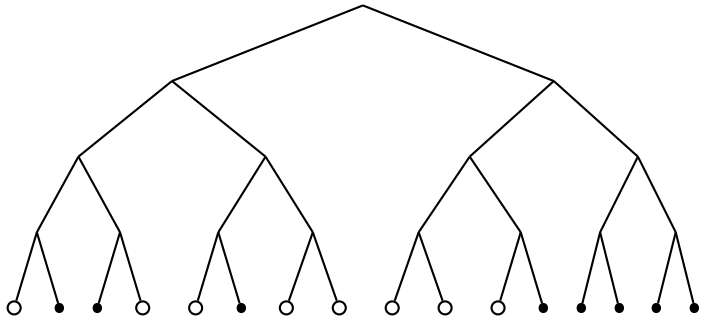


Figura 3.7: Primeros cuatro niveles del árbol de computación asociado una MTP(x)

Ejemplo 9.

En la figura 3.7, se muestran los primeros cuatro niveles del árbol de computación obtenido al aplicar una MTP a una entrada x. Las hojas representadas por círculos blancos corresponden a computaciones de aceptación, y las hojas representadas por círculos negros corresponden a las computaciones de rechazo; en la figura hay 8 computaciones de rechazo y 8 computaciones de aceptación.

Ahora se considerá la interpretación de las hojas en el árbol de computación asociado a una entrada x. Obsérvese que tales hojas ya no denotan pasos no determinísticos, sino pasos aleatorios; el siguiente estado depende de la instrucción que se seleccione aleatoriamente, es decir, si la cabeza de lectura- escritura está en la situación (q, s) y $\delta(q, s) = (q_Y, s', d)$ o $\delta(q, s) = (q_1, s', d)$; el siguiente estado depende de la elección aleatoria de la instrucción a seguir. Una máquina de Turing no determinística acepta una entrada x si en el árbol de computación asociado a la $MTN(x)$ existe un camino de computación de aceptación, la aceptación de una MTP para una entrada x podría depender del número de caminos de computaciones de aceptación en el árbol de computación asociado con la máquina, es decir la probabilidad de obtener una computación de aceptación.

Definición 3.14. *Dada una máquina de Turing probabilística(MTP) y una entrada x se define la razón de aceptación, $\alpha(MTP, x)$ como la razón entre el número de computaciones de aceptación en el árbol de computación asociado a $MTP(x)$ sobre el número total de computaciones de la máquina.*

Definición 3.15. *Dada una máquina de Turing probabilística(MTP) y una entrada x se*

define la razón de rechazo, $\beta(MTP, x)$ como la razón entre el número de computaciones de rechazo en el árbol de computación asociado a $MTP(x)$ sobre el número total de computaciones de la MTP.

Teniendo en cuenta las definiciones anteriores, se define el error de probabilidad en las máquinas de Turing probabilísticas.

Definición 3.16. *Dada una entrada x, el error de probabilidad de una MTP se define como la razón de rechazo, $\beta(MTP, x)$, si $MTP(x)$ acepta la entrada ; o como la razón de aceptación, $\alpha(MTP, x)$, si $MTP(x)$ rechaza.*

El criterio de aceptación de las MTP será presentado en las siguientes secciones.

Con base en el criterio de aceptación se clasifican las máquinas de Turing probabilísticas en :

- PP-máquinas
- BPP-máquinas
- R-máquinas
- ZPP-máquinas

Cada una de estas clases de máquinas serán estudiadas, para definir las diferentes clases de complejidad probabilística que se conocen.

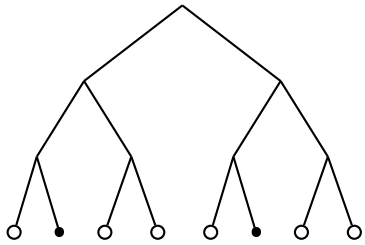
3.1.1. PP-máquinas

Definición 3.17. *Una máquina de Turing probabilística MTP se dice que es una PP-máquina si:*

1. *Para cualquier entrada x, $MTP(x)$ acepta si $\alpha(MTP, x) > \frac{1}{2}$.*
2. *Para cualquier entrada x, $MTP(x)$ rechaza si $\beta(MTP, x) \geq \frac{1}{2}$.*

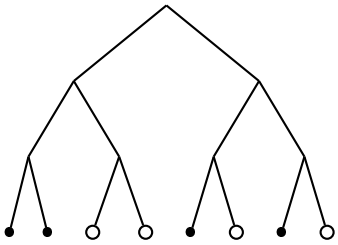
Ejemplo 10.

Se aplica una PP-máquina a una entrada x.



$\alpha(MTP, x) = \frac{3}{4}$, $MTP(x)$ acepta

(a)



$\beta(MTP, x) = \frac{1}{2}$, $MTP(x)$ rechaza

(b)

Figura 3.8: Una PP-máquina

- Si se obtiene el árbol de computación de la figura 3.8 (a), la PP-máquina acepta la entrada x, porque $\alpha(MTP, x) = \frac{3}{4} \geq \frac{1}{2}$.
- Si se obtiene el árbol de computación de la figura 3.8 (b), la PP-máquina rechaza la entrada, porque $\beta(MTP, x) = \frac{1}{2}$.

Ejemplo 11.

Se mostrararán ejemplos de problemas que pueden ser resueltos por una PP-máquina

- **$\frac{1}{2}$ -Satisfabilidad**

  Entrada: Una fórmula ϕ en forma normal conjuntiva, sobre un conjunto U de variables.

  Pregunta: ¿Es ϕ satisfactible por más de $\frac{1}{2}$ de las posibles asignaciones de verdad de sus variables?.

  Si $|U| = n$ se tienen 2^n posibles asignaciones de valores de verdad para las variables en U; se debe verificar si al menos $2^{n-1} + 1$ de estas asignaciones son satisfacientes para la formula ϕ.

- Recuerde del teorema (1.3.1) que el algoritmo Random Walks (RW), cuando es aplicado a instancias satisfacientes de $2SAT$, la probabilidad de que una asignación de verdad satisfaciente sea encontrada es de al menos $\frac{1}{2}$, es decir, $\alpha(RW, \phi) \geq \frac{1}{2}$ si la formula ϕ es satisfactible. Sean L el lenguaje para $2SAT$ definido por RW y ϕ una entrada de $2SAT$. Por el teorema (1.3.1) se tiene que:

  - si $x \in L$, entonces $\alpha(RW, x) > \frac{1}{2}$ ó

- si $x \notin L$, entonces $\beta(RW, x) = 1$

Por lo tanto, el lenguaje L para $2SAT$ es decidido por una PP-máquina.

Desde un punto de vista práctico, las PP-máquinas corresponden a los algoritmos probabilísticos menos útiles, por ejemplo si L es un lenguaje decidido por una PP-máquina, entonces una entrada x podría estar en L con probabilidad de aceptación $\alpha(MTP, x) = \frac{1}{2} + 2^{-p(n)}$, con justamente dos caminos de computación de aceptación más que de computaciones de rechazo. No es posible que por medio de una experimentación eficiente, se pueda encontrar una probabilidad de aceptación tan alta como sea posible, es decir, no es posible hacer el error de probabilidad arbitrariamente pequeño con un número polinomial de repeticiones.

Para entender mejor lo anterior, imagine que se tiene una moneda cargada, esto es, uno de sus lados es más probable que aparezca que el otro. Se sabe que un lado tiene probabilidad $\frac{1}{2} + \epsilon$ para algún $\epsilon \in (0, \frac{1}{2})$ y el otro $\frac{1}{2} - \epsilon$, pero no se sabe cual es cual. ¿Cómo se podría saber cual lado es más probable?, una forma de verificar esto podría ser lanzar la moneda muchas veces y escoger el lado que aparezca más veces, para ser el que tiene probabilidad $\frac{1}{2} + \epsilon$. La pregunta ahora es ¿Cuántas veces se debe lanzar la moneda para escoger correctamente y con alta probabilidad?. El siguiente resultado es muy útil para analizar algoritmos aleatorios y será utilizado para probar por qué las PP-máquinas son los algoritmos probabilísticos menos útiles.

LEMA 4 ([Pap94]La cota de Chernoff). *Supóngase que $x_1, x_2, \ldots, x_n$ son variables aleatorias independientes que toman valores 0 y 1 con probabilidades p y $1 - p$, respectivamente, y considere su suma $X = \sum_{i=1}^{n} x_i$. Entonces para todo $0 \leq \theta \leq 1$, $prob[X \geq (1 + \theta)pn] \leq e^{-\frac{\theta^2}{3}pn}$.*

Demostración. Como $x_1, x_2, \ldots, x_n$ son variables aleatorias, $R_x = \{0, 1\}$ con probabilidades p y $1 - p$, respectivamente y $X = \sum_{i=1}^{n} x_i$ donde la probabilidad de la variable aleatoria X tiene una distribución de probabilidad binomial, es decir

$$prob(X = t) = \binom{n}{p} p^t (1-p)^{n-t}$$

Se calculará la esperanza de la variable aleatorio X, como la esperanza de cada una de las variables x_i está dada por, $\varepsilon(x_i) = 0p(x_i = 0) + 1p(x_i = 1) = p$ para cada $i = 1, 2, \ldots, n$. Entonces $\varepsilon(X) = \varepsilon(\sum_{i=1}^{n} x_i) = \sum_{i=1}^{n} \varepsilon(x_i) = \sum_{i=1}^{n} p = np$.

Por otro lado, para cualquier número real positivo t se tiene que:

$$prob[X \geq (1+\theta)pn] = prob[e^{Xt} \geq e^{t(1+\theta)pn}.]$$

Luego, por el lema 1.3.1 se tedría que $prob[X \geq k\varepsilon(X)pn] \leq \frac{1}{k}$ para cualquier $k > 0$, entonces $prob[e^{tX} \geq k\varepsilon(e^{tX})] \leq \frac{1}{k}$ para cualquier $k > 0$. Tomando $k = e^{t(1+\theta)pn}[\varepsilon(e^{tX})]^{-1}$ se obtendría que:

$$prob[e^{Xt} \geq e^{t(1+\theta)pn}[\varepsilon(e^{tX})]^{-1}\varepsilon(e^{tX})] \leq \frac{\varepsilon(e^{tX})}{e^{t(1+\theta)pn}}$$
$$prob[e^{Xt} \geq e^{t(1+\theta)pn}] \leq \frac{\varepsilon(e^{tX})}{e^{t(1+\theta)pn}}.$$

pero,

$$prob[X \geq (1+\theta)pn] = prob[e^{Xt} \geq e^{t(1+\theta)pn}].$$

Entonces

$$prob[X \geq (1+\theta)pn] \leq \frac{\varepsilon(e^{tX})}{e^{t(1+\theta)pn}}.$$

La esperanza de e^{tx_i} está dada por:

$$\varepsilon(e^{tx_i}) = e^0 p(x_i = 0) + e^t p(x_i = 1) = 1 - p + e^t p = 1 + p(e^t - 1).$$

Como $X = \Sigma_{i=1}^{n} x_i$ se tiene que,

$$\varepsilon(e^{tX}) = (\varepsilon(e^{tx_i}))^n = (1 + p(e^t - 1))^n.$$

Luego,

$$prob[X \geq (1+\theta)pn] \leq \frac{(1 + p(e^t - 1))^n}{e^{t(1+\theta)pn}}.$$

Se probará por inducción sobre n que para cualquier número a real positivo, $(1+a)^n \leq e^{an}$

1. Para $n = 1$, se tiene que,

   $e^a = \sum_{k=0}^{\infty} \frac{a^k}{k!} = 1 + a + \frac{a^2}{2!} + \ldots \geq 1 + a$. Así, $(1 + a) \leq e^a$

2. Suponga que la desigualdad se cumple para $n-1$, es decir, para todo número a real positivo $(1 + a)^{n-1} \leq e^{a(n-1)}$ y se probará que el resultado es válida para n

   Sea a un número real positivo, entonces $(1 + a)^n = (1 + a)^{n-1}(1 + a)$, pero por hipótesis inductiva y el caso $n - 1$, se tiene que $(1 + a)^n \leq e^{a(n-1)}e^a = e^{na}$

Por lo tanto $(1 + a)^n \leq e^{na}$, para todo $n \in \mathbb{N}$ y para todo número real positivo a.

Utilizando este resultado se tiene,

$$\begin{aligned} prob[X \geq (1 + \theta)pn] &\leq (1 + p(e^t - 1))^n (e^{-t(1+\theta)pn}) \\ &\leq e^{np(e^t-1)} e^{-t(1+\theta)pn} = e^{np[-(1+\theta)t+(e^t-1)]}. \end{aligned}$$

Tomando $t = \ln(1 + \theta)$ se obtiene,

$$\begin{aligned} &- (1 + \theta)t + (e^t - 1) = -(1 + \theta)\ln(1 + \theta) \ + \ e^{\ln(1+\theta)} - 1 \\ &= -(1 + \theta)\ln(1 + \theta) + 1 + \theta - 1 = \theta - (1 + \theta)\ln(1 + \theta). \end{aligned}$$

Entonces

$$e^{np[-(1+\theta)t+(e^t-1)]} = e^{np(\theta-(1+\theta)\ln(1+\theta))}.$$

Además, se sabe que $\ln(1 + \theta) = \theta - \frac{\theta^2}{2} + \frac{\theta^3}{3}, -\frac{\theta^4}{4} + \frac{\theta^5}{5} - \ldots$

$$\begin{aligned} \theta - (1 + \theta)\ln(1 + \theta) &= \theta - (1 + \theta)(\theta - \frac{\theta^2}{2} + \frac{\theta^3}{3} - \frac{\theta^4}{4} + \frac{\theta^5}{5}, - \ldots) \\ &= \theta - \big[\theta - \frac{\theta^2}{2} + \frac{\theta^3}{3} - \frac{\theta^4}{4} + \frac{\theta^5}{5} - \cdots + \theta^2 - \frac{\theta^3}{2} + \frac{\theta^4}{3} - \frac{\theta^5}{4} + \frac{\theta^6}{5} - \ldots\big] \\ &= \theta - \big[\theta + \frac{\theta^2}{2} - \frac{\theta^3}{6} + \frac{\theta^4}{12} - \frac{\theta^5}{20} + \ldots\big] \\ &= -\frac{\theta^2}{2} + \frac{\theta^3}{6} - \frac{\theta^4}{12} + \frac{\theta^5}{20} + \ldots \end{aligned}$$

Pero,

$$-\frac{\theta^2}{2} + \frac{\theta^3}{6} - \frac{\theta^4}{12} + \frac{\theta^5}{20} + \cdots = \sum_{k=1}^{\infty} \frac{(-1)^k \theta^{k+1}}{k(k+1)}$$

Esta serie es convergente, ya que es una serie alternante y decreciente, es decir

$\lim_{k\to\infty} \frac{\theta^{k+1}}{k(k+1)} = 0$. Como la serie es convergente se pueden agrupar términos y la serie obtenida también es convergente y converge al mismo número. Entonces

$$\begin{aligned}
&\left(-\frac{\theta^2}{2}+\frac{\theta^3}{6}\right)+\left(-\frac{\theta^4}{12}+\frac{\theta^5}{20}\right)+\cdots+\left(\frac{-\theta^{k+1}}{k(k+1)}+\frac{\theta^{k+2}}{(k+1)(k+2)}\right)\cdots \\
&=\sum_{k=1}^{\infty}\frac{-(k+2)\theta^{k+1}+k\theta^{k+2}}{k(k+1)(k+2)}.
\end{aligned}$$

Se tiene que, $\theta^{k+1} \geq \theta^{k+2}$, porque $0 \leq \theta \leq 1$, entonces la serie $\sum_{k=1}^{\infty} a_n$, donde $a_n = \frac{-(n+2)\theta^{n+1}+n\theta^{n+2}}{n(n+1)(n+2)}$ es de términos negativos.

Se analizará el comportamiento de esta serie. Como $\theta \in [0,1]$ es claro que.

$$\begin{aligned}
-1+\theta &\leq 0 \\
-n-3+n+2+n\theta+\theta-n\theta &\leq 0 \\
-(n+3)+(n+2)+\theta(n+1)-n\theta &\leq 0
\end{aligned}$$

$$-(n-3)+\theta(n+1) \leq -(n+2)+n\theta. \tag{3.8}$$

También se tiene que, $n\theta \leq (n+3)$ y multiplicando a ambos lados de la desigualdad anterior por θ^{n+1} se obtiene.

$$n\theta^{n+2} \leq \theta^{n+1}(n+3) \tag{3.9}$$

Multiplicando 3.8 y 3.9 lado a lado.

$$\begin{aligned}
n\theta^{n+2}(-(n+3)+\theta(n+1)) &\leq \theta^{n+1}(n+3)(-(n+2)+n\theta) \\
n[-(n+3)\theta^{n+2}+\theta^{n+3}(n+1)] &\leq (n+3)[-(n+2)\theta^{n+1}+n\theta^{n+2}] \\
\frac{-(n+3)\theta^{n+2}+\theta^{n+3}(n+1)}{n+3} &\leq \frac{-(n+2)\theta^{n+1}+n\theta^{n+2}}{n} \\
\frac{-(n+3)\theta^{n+2}+\theta^{n+3}(n+1)}{(n+1)(n+2)(n+3)} &\leq \frac{-(n+2)\theta^{n+1}+n\theta^{n+2}}{n(n+1)(n+2)}
\end{aligned}$$

Esto es, $a_{n+1} \leq a_n$ para toda $n \in \mathbb{N}$, es decir, los términos de la serie son decrecientes, luego la serie puede ser acotada con el primer término de dicha serie. Así

$$-\frac{\theta^2}{2}+\frac{\theta^3}{6}-\frac{\theta^4}{12}+\cdots\leq-\frac{\theta^2}{2}+\frac{\theta^3}{6}=\frac{\theta^2(-3+\theta)}{6}$$

Pero,

$$3-\theta\geq 2\Leftrightarrow -\theta^2(3-\theta)\leq -2\theta^2\Leftrightarrow\frac{-\theta^2(3-\theta)}{6}\leq\frac{-2\theta^2}{6}\Leftrightarrow\frac{-\theta^2(3-\theta)}{6}\leq-\frac{\theta^2}{3}$$

$$\Rightarrow-\frac{\theta^2(3-\theta)}{6}np\leq-\frac{\theta^2}{3}np\Rightarrow e^{-\frac{\theta^2(3-\theta)}{6}np}\leq e^{-\frac{\theta^2}{3}np}$$

Como $prob[X\geq(1+\theta)pn]\leq e^{-\frac{\theta^2(3-\theta)}{6}np}$ y $e^{-\frac{\theta^2(3-\theta)}{6}np}\leq e^{-\frac{\theta^2}{3}np}$.
Entonces $prob[X\geq(1+\theta)pn]\leq e^{-\frac{\theta^2}{3}np}$ $\diamondsuit$

Del lema 4 se tiene que, la probabilidad de que una variable aleatoria de bernoulli, con distribución de probabilidad binomial, tome valores menores que la esperanza por un factor positivo decrece exponencialmente, es decir la probabilidad de que una variable aleatoria se desvíe de su esperanza decrece exponencialmente con la desviación.

Corolario 3.3. *[Pap94] Sean $x_1,x_2,\ldots,x_n$ variables aleatorias que toman valores 0 y 1, y $p=\frac{1}{2}+\epsilon$ para algún $\epsilon>0$, entonces la probabilidad de que $X=\sum_{i=1}^{n}x_i\leq\frac{n}{2}$ es a lo más $e^{-\frac{\epsilon^2 n}{6}}$.*

Demostración. Del lema 4 se tiene que, $prob[X\geq(1+\theta)pn]\leq e^{-\frac{\theta^2}{3}pn}$.
Tomando $\theta=\frac{\epsilon}{\frac{1}{2}+\epsilon}$, se tendría que $prob[X\geq(1+\theta)pn]=prob[X\geq\frac{n}{2}(1+4\epsilon)]$ y $e^{-\frac{\theta^2}{3}pn}=e^{-\frac{2\epsilon^2 n}{3(1+2\epsilon)}}$. Entonces, $prob[X\geq\frac{n}{2}(1+4\epsilon)]\leq e^{-\frac{2\epsilon^2 n}{3(1+2\epsilon)}}$.
De ahí, $prob[X<\frac{n}{2}(1+4\epsilon)]\leq 1-e^{-\frac{2\epsilon^2 n}{3(1+2\epsilon)}}$, por el complemento de la probabilidad

Tomando $\epsilon\in\left(\frac{6\ln 2-\sqrt{36\ln^2 2+24n\ln 2}}{4n},\frac{6\ln 2+\sqrt{36\ln^2 2+24n\ln 2}}{4n}\right)$ se tiene que,
$1-e^{-\frac{2\epsilon^2 n}{3(1+2\epsilon)}}<e^{-\frac{2\epsilon^2 n}{3(1+2\epsilon)}}$. Así, $prob[X<\frac{n}{2}(1+4\epsilon)]<e^{-\frac{2\epsilon^2 n}{3(1+2\epsilon)}}$.

Por otro lado, considere $0 \leq \epsilon \leq \frac{3}{2}$. Luego

$$
\begin{aligned}
&2\epsilon - 3 \leq 0 \quad \text{como } 3\epsilon^2 n \geq 0 \text{ entonces.} \\
&3\epsilon^2 n(2\epsilon - 3) \leq 0 \Leftrightarrow 6\epsilon^3 n - 9\epsilon^2 n \leq 0 \\
&\Leftrightarrow 3\epsilon^2 n + 6\epsilon^3 n - 12\epsilon^2 n \leq 0 \Leftrightarrow 3\epsilon^2 n + 6\epsilon^3 n \leq 12\epsilon^2 n \\
&\Leftrightarrow 3\epsilon^2 n(1 + 2\epsilon) \leq 12\epsilon^2 n \Leftrightarrow 3\epsilon^2 n \leq \frac{12\epsilon^2 n}{1 + 2\epsilon} \\
&\Leftrightarrow \frac{3\epsilon^2 n}{24} \leq \frac{12\epsilon^2 n}{24(1 + 2\epsilon)} \Leftrightarrow \frac{\epsilon^2 n}{6} \leq \frac{2\epsilon^2 n}{3(1 + 2\epsilon)} \\
&\Leftrightarrow e^{\frac{\epsilon^2 n}{6}} \leq e^{\frac{2\epsilon^2 n}{3(1+2\epsilon)}} \Leftrightarrow e^{\frac{-2\epsilon^2 n}{3(1+2\epsilon)}} \leq e^{\frac{-\epsilon^2 n}{6}}
\end{aligned}
$$

Así, $prob[X < \frac{n}{2}(1 + 4\epsilon)] \leq e^{\frac{-\epsilon^2 n}{6}}$. Como $prob[X \leq \frac{n}{2}] \leq prob[X < \frac{n}{2}(1 + 4\epsilon)] \leq e^{\frac{-\epsilon^2 n}{6}}$, se tiene que $prob[X \leq \frac{n}{2}] \leq e^{\frac{-\epsilon^2 n}{6}}$ $\diamondsuit$

Continuando con el análisis de la eficiencia de las PP-máquinas, suponga que $p = \alpha(MPT,\, x) = \frac{1}{2} + 2^{-p(n)}$, luego $\epsilon = 2^{-p(n)}$. Dado que la variable aleatoria X en este caso representa el número de computaciones de rechazo en n ensayos independientes, por el corolario se tiene que.

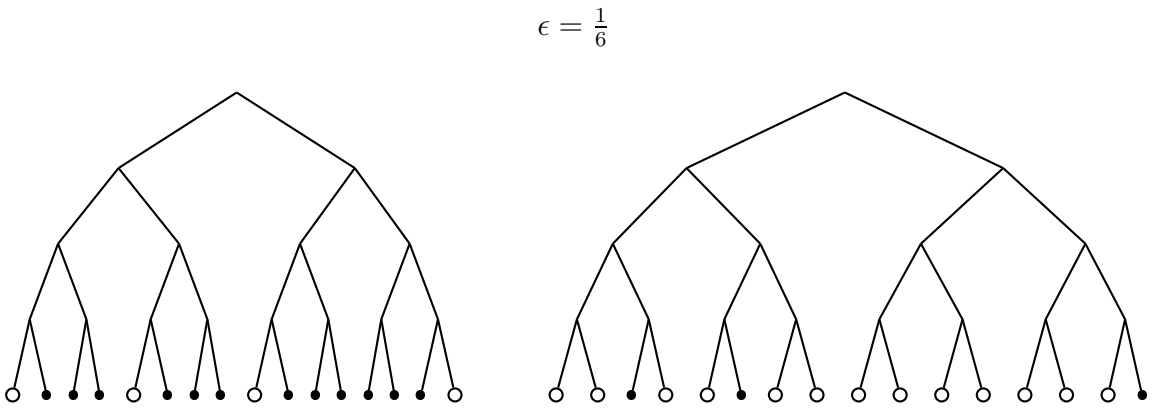
$$prob[X \leq \frac{n}{2}] \leq e^{-\frac{(2^{-p(n)})^2 n}{3}} \Leftrightarrow prob[X \leq \frac{n}{2}] \leq e^{-\frac{n}{6 * 2^{2p(n)}}}$$

Es decir, se necesitan un número exponencial de ensayos ($2^{2p(n)}$), para hacer el error de probabilidad arbitrariamente pequeño.

3.1.2. BPP-máquinas

Definición 3.18. *Una máquina de Turing probabilística MTP se dice que es una BPP-máquina si existe una constante $\epsilon \in (0, \frac{1}{2})$ tal que.*

1. *Para cualquier x, se cumple cualquiera de los casos: $\alpha(MTP, x) > \frac{1}{2} + \epsilon$ ó $\beta(MTP, x) > \frac{1}{2} + \epsilon$.*
2. *Para cualquier x, $MTP(x)$ acepta si $\alpha(MTP, x) > \frac{1}{2} + \epsilon$.*
3. *Para cualquier x, $MTP(x)$ rechaza si $\beta(MTP, x) > \frac{1}{2} + \epsilon$.*



$\beta(MTP, x) = \frac{3}{4}$, $MTP(x)$ rechaza (a)

$\alpha(MTP, x) = \frac{13}{16}$, $MTP(x)$ acepta (b)

Figura 3.9: Una BPP-máquina

Ejemplo 12.

Aplicando una BPP-máquina a una entrada x.

- Si se obtiene el árbol de computación de la figura 3.9 (a), la BPP-máquina rechaza la entrada x, porque existe $\varepsilon = \frac{1}{6}$ tal que $\beta(MTP, x) = \frac{12}{16} = \frac{3}{4} > \frac{2}{3} = \frac{1}{2} + \varepsilon$.
- Si se tiene el árbol de computación de la figura 3.9 (b), la BPP-máquina acepta la entrada x, porque existe $\varepsilon = \frac{1}{6}$ tal que $\alpha(MTP, x) = \frac{13}{16} > \frac{2}{3} = \frac{1}{2} + \varepsilon$.

Dada una entrada x si $MTP(x)$ acepta, el error de probabilidad de la $MTP(x)$ es igual a $\beta(MTP, x)$, luego $\alpha(MTP, x) > \frac{1}{2} + \epsilon$ para algún $\epsilon \in (0, \frac{1}{2})$, así

$$\beta(MTP, x) < 1 - \frac{1}{2} - \epsilon, \text{entonces}$$
$$\beta(MTP, x) < \frac{1}{2} - \epsilon \quad \text{para algún} \quad \epsilon \in (0, \frac{1}{2}).$$

Luego, el error de probabilidad de la $MTP(x)$ está acotado "lejos"de $\frac{1}{2}$; por la definición de las BPP-máquinas note que en ella se incluyen problemas con error de probabilidad muy proximos a $\frac{1}{2}$, como también problemas con error de probabilidad bajo, por ejemplo 0.0000000001.

Ejemplos de problemas que pueden ser decididos por una BPP-máquina son los problemas presentados en las secciones 1.3 y 2.1, ya que todos ellos tienen algoritmos probabilísticos con error de probabilidad acotado. Es decir, si el algoritmo acepta la entrada, este acepta con probabilidad de al menos $\frac{1}{2}$, esto es $\alpha(a,x) > \frac{1}{2}$, donde a es el respectivo algoritmo. Además si el algoritmo rechaza la entrada la probabilidad de error es cero, esto es, $\beta(a,x) = 1$.

Las BPP-máquinas no son más potentes que las PP-máquinas. En efecto si un lenguaje L es decidido por una BPP-máquina, entonces la misma máquina interpretada como una PP-máquina decide L, este resultado se probará más adelante.

La razón para definir las BPP-máquinas de esta manera es que si un algoritmo tiene error de probabilidad acotado, éste se puede hacer arbitrariamente pequeño simplemente repitiendo el algoritmo un número polinomial de veces; al contrario de lo que ocurre con las PP-máquinas, que se necesitan un número exponencial de repeticiones para hacer el error de probabilidad arbitrariamente pequeño, este resultado se probará en el siguiente teorema.

Teorema 3.1.1. *[PC94] Sean MP una BPP-máquina y q un polinomio. Entonces existe una máquina de Turing probabilística MP' tal que para cualquier x con $|x| = n$, si $MP(x)$ acepta entonces $\alpha(MP', x) > 1 - 2^{-q(n)}$ y si $MP(x)$ rechaza entonces $\beta(MP', x) > 1 - 2^{-q(n)}$.*

Demostración. Por hipótesis MP es una BPP-máquina, se sigue que existe $\epsilon \in (0, \frac{1}{2})$ tal que para cualquier x, $MP(x)$ acepta siempre que $\alpha(MP, x) > \frac{1}{2} + \epsilon$ y $MP(x)$ rechaza si $\beta(MP, x) > \frac{1}{2} + \epsilon$.
Sea t un número impar que será especificado después. La máquina de Turing probabilística MP' es la siguiente.

inicio{entrada: x}
  $cont \leftarrow 0$
  para $i \leftarrow 1$ **hasta** t **hacer**

inicio

ejecuta $MP(x)$

si $MP(x)$ acepta **entonces**

$cont \leftarrow cont + 1$

fin

si $cont > \frac{t}{2}$ **entonces** acepte

sino rechaza

fin.

Si $MP(x)$ acepta, la razón del número de caminos de computación de MP', tales que el valor de *cont* sea exactamente i, con $i \leq \frac{t}{2}$ sobre el número total de computaciones de la MP', está dado por:[29]

$$\binom{t}{i}\left(\frac{1}{2}+\epsilon\right)^{i}\left(\frac{1}{2}-\epsilon\right)^{t-i}$$

Por otro lado, $\frac{1}{2}+\epsilon \geq \frac{1}{2}-\epsilon$ para todo $\epsilon \in (0, \frac{1}{2})$ y además $\frac{1}{2}-\epsilon \neq 0$, luego $\frac{\frac{1}{2}+\epsilon}{\frac{1}{2}-\epsilon} \geq 1$. De ahí que $\left(\frac{\frac{1}{2}+\epsilon}{\frac{1}{2}-\epsilon}\right)^{\frac{t}{2}-i} \geq 1$.

$$\begin{aligned}
\binom{t}{i}\left(\frac{1}{2}+\epsilon\right)^{i}\left(\frac{1}{2}-\epsilon\right)^{t-i} &\leq \binom{t}{i}\left(\frac{1}{2}+\epsilon\right)^{i}\left(\frac{1}{2}-\epsilon\right)^{t-i}\left(\frac{\frac{1}{2}+\epsilon}{\frac{1}{2}-\epsilon}\right)^{\frac{t}{2}-i} \\
&= \binom{t}{i}\left(\frac{1}{2}+\epsilon\right)^{\frac{t}{2}}\left(\frac{1}{2}-\epsilon\right)^{\frac{t}{2}} \\
&= \binom{t}{i}\left(\left(\frac{1}{2}+\epsilon\right)\left(\frac{1}{2}-\epsilon\right)\right)^{\frac{t}{2}} \\
&= \binom{t}{i}\left(\frac{1}{4}-\epsilon^{2}\right)^{\frac{t}{2}}
\end{aligned}$$

[29]Si la probabilidad de que un evento ocurra es al menos p. La probabilidad de que tal evento ocurra exactamente i veces en t ensayos independientes es a lo más, $\binom{t}{i}p^{i}(1-p)^{t-i}$, lo cual se conoce como **distribución de probabilidad binomial** [Pa71]

Se sabe que MP' acepta si $cont > \frac{t}{2}$, luego se tendría que $MP'(x)$ rechaza la entrada x, si $cont < \frac{t}{2}$. Así, el número de computaciones de rechazo es a lo más $\frac{t-1}{2}$, porque.

$$\begin{aligned} t-2 \ &< t-1 \ < t \\ \frac{t}{2}-1 \ &< \frac{t-1}{2} \ < \frac{t}{2} \end{aligned}$$

Es decir, $\frac{t-1}{2}$ es el menor número entero entre $\frac{t}{2}-1$ y $\frac{t}{2}$

De ahí que la proporción de el número de caminos de computaciones de rechazo de la MP' sobre el número total de computaciones esta dado por:

$$\sum_{i=0}^{\frac{(t-1)}{2}} \binom{t}{i} \left(\frac{1}{2}+\epsilon\right)^{i} \left(\frac{1}{2}-\epsilon\right)^{t-i} \leq \sum_{i=0}^{\frac{(t-1)}{2}} \binom{t}{i} \left(\frac{1}{4}-\epsilon^2\right)^{\frac{t}{2}}.$$

Como $\sum_{i=0}^{t} \binom{t}{i} = \sum_{i=0}^{\frac{t-1}{2}} \binom{t}{i} + \sum_{i=\frac{t+1}{2}}^{t} \binom{t}{i}$ y además $\binom{t}{\frac{t-1}{2}}$ y $\binom{t}{\frac{t+1}{2}}$ son los coeficientes más grandes en las sumatorias porque t es impar y el coeficiente más grande es la mitad del número de términos[30], se sabe que la suma de los coeficientes del binomio es 2^t, es decir $\sum_{i=0}^{t} \binom{t}{i} = 2^t$, así que $\sum_{i=0}^{\frac{t-1}{2}} \binom{t}{i} = \frac{2^t}{2} = 2^{t-1}$.
Ahora,

$$\sum_{i=0}^{\frac{(t-1)}{2}} \binom{t}{i} \left(\frac{1}{4}-\epsilon^2\right)^{\frac{t}{2}} = 2^{t-1}\left(\frac{1}{4}-\epsilon^2\right)^{\frac{t}{2}}$$

$$2^{t-1}\frac{(1-4\epsilon^2)^{\frac{t}{2}}}{2^t} = 2^{-1}(1-4\epsilon^2)^{\frac{t}{2}}$$

Así, el número de caminos de computaciones de rechazo de la $MP'(x)$ sobre el número total de computaciones es a lo más, $\frac{1}{2}(1-4\epsilon^2)^{\frac{t}{2}}$. Luego, se tiene que la probabilidad de aceptación de la MP' esta dada por:

$$1-\frac{1}{2}(1-4\epsilon^2)^{\frac{t}{2}}$$

[30]Esto se puede ver en el triángulo de pascal

Para completar la desmostración del teorema, es suficiente tomar t tal que

$$1 - \frac{1}{2}(1-4\epsilon^2)^{\frac{t}{2}} \geq 1 - 2^{-q(n)}$$

lo cual se cumple si

$$\begin{aligned}
\frac{1}{2}(1-4\epsilon^2)^{\frac{t}{2}} &\leq 2^{-q(n)} = \frac{1}{2^{q(n)}} \\
\frac{1}{2}(1-4\epsilon^2)^{\frac{t}{2}} &\leq \frac{1}{2^{q(n)}} \\
(1-4\epsilon^2)^{\frac{t}{2}} &\leq \frac{2}{2^{q(n)}} \\
2^{q(n)} &\leq 2.\frac{1}{(1-4\epsilon^2)^{\frac{t}{2}}} = 2\left(\frac{1}{1-4\epsilon^2}\right)^{\frac{t}{2}}.
\end{aligned}$$

Aplicando logaritmo base 2 a ambos lados de la desigualdad anterior, se tiene que

$$\begin{aligned}
\lg(2^{q(n)}) &\leq \lg\left[2\left(\frac{1}{1-4\epsilon^2}\right)^{\frac{t}{2}}\right] \\
q(n) &\leq \lg 2 + \lg\left(\frac{1}{1-4\epsilon^2}\right)^{\frac{t}{2}} \\
q(n) &\leq 1 + \frac{t}{2}\lg\left(\frac{1}{1-4\epsilon^2}\right) \\
2(q(n)-1) &\leq t\lg\left(\frac{1}{1-4\epsilon^2}\right) \\
t &\geq \frac{2(q(n)-1)}{\lg\left(\frac{1}{1-4\epsilon^2}\right)}.
\end{aligned}$$

De forma similar, se puede analizar el caso en el que $MP(x)$ rechaza. Así las t repeticiones de la MP' son capaces de simular MP en tiempo polinomial. $\diamondsuit$

Del teorema se tiene que las BPP-máquinas pueden tener error de probabilidad arbitrariamente pequeño (exponencialmente pequeño).

3.1.3. R-máquinas

El test de composición presentado en la sección 2.1.2 es una BPP-máquina. Recuerde que si el algoritmo rechaza, con seguridad n no es primo, pero si el algoritmo acepta, entonces

se establece que n es primo con probabilidad al menos $\frac{1}{2}$. Observe que este algoritmo tiene una característica importante, si la entrada es un número compuesto, la probabilidad de error es cero. Lo anterior lleva a definir la versión de "un lado" de la BPP-máquina, las R-máquinas.

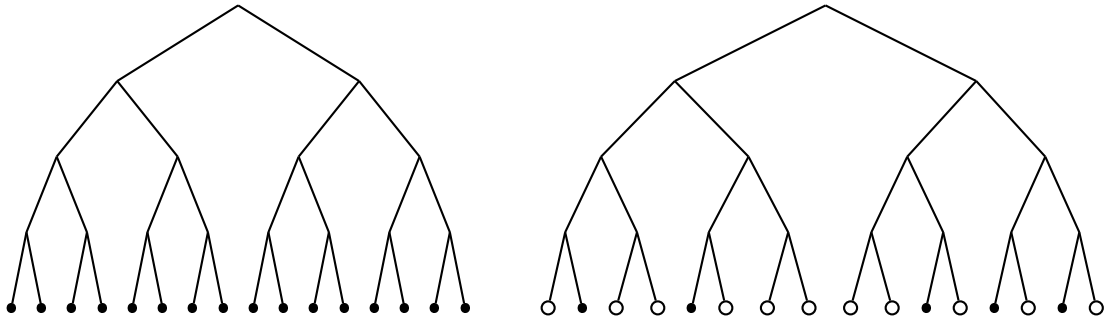
Definición 3.19. *Una máquina de Turing probabilístca MTP se dice que es una R-máquina si:*

1. *Para cualquier x, $\alpha(MTP, x) > \frac{1}{2}$ ó $\beta(MTP, x) = 1$.*
2. *Para cualquier x, $MTP(x)$ acepta si $\alpha(MTP, x) > \frac{1}{2}$.*
3. *Para cualquier x, $MTP(x)$ rechaza si $\beta(MTP, x) = 1$.*

Ejemplo 13.
En la figura 3.10 se muestra el árbol de computación de una R-máquina para una entrada x.

- Si se tiene el árbol de computación de la izquierda (a), en el cual todos los caminos de computación terminan en computaciones de rechazo, es decir $\beta(MTP, x) = 1$, entonces $MTP(x)$ rechaza.
- Si se tiene el árbol de computación de la derecha (b), en el cual, 11 caminos de computación termina en computaciones de aceptación, es decir $\alpha(MTP, x) = \frac{11}{16}$, entonces $MTP(x)$ acepta.

En el caso de las R-máquinas, el error de probabilidad es cero para rechazar entradas. Como ejemplos de problemas que pueden ser resueltos por una R-máquina, son los problemas presentados en la sección 1.3. Por ejemplo, considere nuevamente el algoritmo Random Walks sección 1.3.1, cuando la entrada a este algoritmo es una cláusula no satisfaciente el algoritmo rechaza la entrada para todas la ejecuciones, es decir, $\beta(RW, x) = 1$, pero si la entrada es una cláusula satisfaciente de $2SAT$, el algoritmo acepta con una probabilidad de al menos $\frac{1}{2}$, esto es $\alpha(RW, x) > \frac{1}{2}$. De igual forma se hace el análisis para el problema del cero polinomial y el problema del matching perfecto.



$\beta(MTP, x) = 1$, $MTP(x)$ rechaza (a)

$\alpha(MTP, x) = \frac{11}{16}$, $MTP(x)$ acepta (b)

Figura 3.10: Una R-máquina

El siguiente teorema establece que las R-máquinas no son más potentes que las BPP-máquinas.

Teorema 3.1.2. *[PC94] Dado un lenguaje L decidido por una R-máquina MP, existe una BPP-máquina MP' que decide L.*

Demostración. Por hipótesis, MP es una R-máquina para cada entrada x, $MP(x)$ acepta siempre que $\alpha(MP, x) > \frac{1}{2}$, y $MP(x)$ rechaza siempre que $\beta(MP, x) = 1$.

Sea MP' una BPP-máquina, la cual simula 2 veces MP y se muestra a continuación.

inicio{entrada: x}
  repetir 2 veces
    ejecuta $MP(x)$
    si $MP(x)$ acepta **entonces**
      "acepta"
  "rechaza"
fin.

La máquina MP' ejecuta MP dos veces y cualquier computación de ella acepta si al menos una computación de $MP(x)$ acepta.

La máquina $MP'(x)$ acepta, siempre que $\alpha(MP', x) > 1 - \frac{1}{4}$, puesto que $MP(x)$ acepta si $\alpha(MP, x) > \frac{1}{2}$. De ahí, $\beta(MP, x) < \frac{1}{2}$, pero MP' ejecuta dos veces MP, luego $\beta(MP, x) < \frac{1}{4}$, esto es la probabilidad de error de la MP es menor que $\frac{1}{4}$. De lo anterior, se tiene que la probabilidad de error de la $MP'(x)$, dado que $MP(x)$ acepte, es menor que $\frac{1}{4}$, es decir, la probabilidad de aceptación de la $MP'(x)$ es $\alpha(MP', x) > 1 - \frac{1}{4}$.

Ahora, si $MP(x)$ rechaza, entonces ningún camino de computación de la $MP'(x)$ asociado a la entrada x acepta. Así, $\beta(MP, x) = 1$.

Como $1 - \frac{1}{4} = \frac{1}{2} + \frac{1}{4}$, tomando $\epsilon = \frac{1}{4}$, se tiene que $\alpha(MP', x) > \frac{1}{2} + \epsilon$ y además $MP'(x)$ acepta, si $MP(x)$ acepta.

Entonces, existe $\epsilon = \frac{1}{4} \in (0, \frac{1}{2})$ tal que $MP'(x)$ acepta si $\alpha(MP', x) > \frac{1}{2} + \epsilon$, esto es, se ha construido una BPP-máquina que decide L $\diamondsuit$

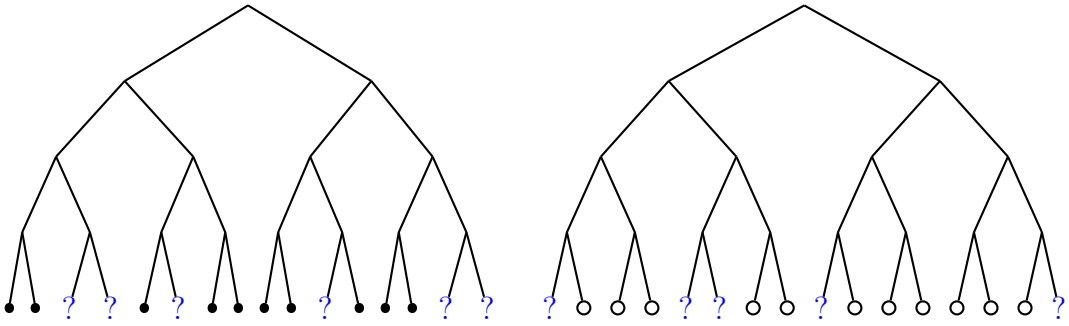
Del teorema se tiene que tanto el error de probabilidad de las R-máquinas como el error de probabilidad de las BPP-máquinas, puede hacerse arbitrariamente pequeño en tiempo polinomial, por el teorema 3.1.1

3.1.4. ZPP-máquinas

En las diferentes clases de máquinas presentadas hasta el momento la probabilidad de error puede hacerse arbitrariamente pequeña, es decir estas máquinas pueden dar respuestas equivocadas con baja probabilidad. Es por esto que son útilizadas para formalizar los algoritmos de Monte Carlo, los cuales devuelven respuestas equivocadas (mienten) con baja probabilidad; la última clase de máquinas de Turing que se presentará es la clase ZPP, la cual es utilizada para formalizar los algoritmos de las Vegas.

Definición 3.20. *Una máquina de Turing probabilística MTP es una ZPP-máquina si:*

1. *Existe un nuevo estado de parada agregado a los ya conocidos de aceptación y rechazo, el estado "no sé", es decir el conjunto de estados para la MTP está formado al menos por los estados $\{q_Y, q_N, stop, no\ sé\}$*



$\beta(MTP, x) = \frac{10}{16}$, $MTP(x)$ rechaza $\quad$ $\alpha(MTP, x) = \frac{11}{16}$, $MTP(x)$ acepta

(a) $\quad$ (b)

Figura 3.11: Una ZPP-máquina

2. *Para cualquier x, se tiene alguno de los dos casos:* $\alpha(MTP, x) > \frac{1}{2} \wedge \beta(MTP, x) = 0$ *ó* $\beta(MTP, x) > \frac{1}{2} \wedge \alpha(MTP, x) = 0$

3. *Para cualquier x, $MTP(x)$ acepta si* $\alpha(MTP, x) > \frac{1}{2}$

4. *Para cualquier x, $MTP(x)$ rechaza si* $\beta(MTP, x) > \frac{1}{2}$

Ejemplo 14.

En la figura 3.11 se tienen las computaciones de un ZPP-máquina asociada con alguna entrada x.

- Si se obtiene el árbol de computación de la izquierda (a), la ZPP-máquina rechaza la entrada, porque $\beta(MTP, x) = \frac{10}{16} > \frac{1}{2}$.
- Si se obtiene el árbol de computación de la derecha (b), la ZPP-máquina acepta la entrada, porque $\alpha(MTP, x) = \frac{11}{16} > \frac{1}{2}$.

Las computaciones que terminan en el estado "no sé" son representados por el signo de pregunta (?).

Así, el árbol de computación de un ZPP-máquina asociado a una entrada x, no contiene caminos de computación contrarios, es decir caminos de computación que terminen en computaciones de aceptación y otros en computaciones de rechazo. Esto es, en la relación δ para esta clase de máquinas las computaciones pueden parar en:

- Sólo en alguno de los estados $q_Y, q_N, stop$.
- o en el estado "no sé"

Es decir, estas máquinas no dan respuestas equivocadas.

Ejemplo 15.

Se mostrarán ejemplos de problemas que pueden ser resueltos por una ZPP-máquina. Recuerde de la sección 2.2 los algoritmos presentados para los problemas de ordenar un vector y encontrar la factorización de un entero grande.

- Para el problema de ordenar un vector, se mostró el algoritmo Quicksort aleatorio, este algoritmo siempre ordena un vector V con n componentes, en un tiempo esperado de $n \lg n$.
- Para el problema de la factorización de enteros grandes, se mostró un algoritmo que siempre encuentra la factorización correcta y tiene probabilidad mayor que $\frac{1}{2}$ de tomar las decisiones correctas.

El siguiente teorema muestra que las ZPP-máquinas no son más potentes que las R-máquinas.

Teorema 3.1.3. *[PC94] Dado un lenguaje L decidido por una ZPP-máquina MP, existe una R-máquina MP' que decide L.*

Demostración. Como L es un lenguaje decidido por una ZPP-máquina. Para cualquier x, la máquina $MP(x)$ acepta si $\alpha(MP, x) > \frac{1}{2} \ \wedge \ \beta(MP, x) = 0$ o la máquina $MP(x)$ rechaza si $\beta(MP, x) > \frac{1}{2} \ \wedge \ \alpha(MP, x) = 0$.

Supóngase que MP' es la máquina de Turing probabilística que ejecuta MP tal que $MP'(x)$ acepta si, y sólo si, $MP(x)$ acepta.

Si $MP(x)$ rechaza la entrada, se tiene que $\beta(MP, x) > \frac{1}{2}$ y $\alpha(MP, x) = 0$, de ahí que para todas las computaciones de la máquina MP' se tiene que $\alpha(MP', x) = 0$ así,

$\beta(MP', x) = 1$, esto es, MP' rechaza si $\beta(MP', x) = 1$.

Si $MP(x)$ acepta, se tiene que $\alpha(MP, x) > \frac{1}{2}$, entonces $\alpha(MP', x) > \frac{1}{2}$, es decir $MP'(x)$ acepta la entrada x si $\alpha(MP', x) > \frac{1}{2}$.

Por lo tanto, $MP'(x)$ acepta si $\alpha(MP', x) > \frac{1}{2}$ y rechaza si $\beta(MP', x) = 1$. Esto es, existe una R-máquina que decide L. $\diamondsuit$

3.2. CLASES DE COMPLEJIDAD PROBABILÍSTICA

A pesar de que las máquinas de Turing probabilísticas: BPP-máquinas, R-máquinas, ZPP-máquinas y PP-máquinas, realizan escogencias aleatorias, hay una gran diferencia entre las tres primeras y la última. Las tres primeras son posibles nociones de algoritmos aleatorios eficientes en la práctica, mientras que la cuarta no, lo cual se garantiza en la sección 3.1.1; a cada máquina de Turing probabilística se asocia un lenguaje, que es precisamente el lenguaje decidido por dicha máquina. La clase de complejidad definida por estos tipos de máquinas corresponde al conjunto de todos los lenguajes que son decididos por tales máquina.

La clase PP fue definida en 1972 por Jhon Gill e independientemente por Janos Simon en 1974.

Definición 3.21. *La clase PP es el conjunto de todos los lenguajes que son decididos por una PP-máquina, es decir.*

$$PP = \{L : L \text{ es un lenguaje y existe una } PP\text{-máquina } MP \text{ tal que } L = L(MP)\}$$

De la definición anterior se tiene que un lenguaje $L \in PP$ si existe una máquina de Turing probabilística MP tal que para toda x, $x \in L$ si, y sólo si, más de la mitad de las computaciones de $MP(x)$ paran en aceptación, en este caso se dice que $MP(x)$ decide L por "mayoría".

Definición 3.22. *La clase BPP es el conjunto de todos los lenguajes que son decididos por una BPP-máquina, es decir.*

$$BPP = \{L : L \text{ es un lenguaje y existe una } BPP\text{-máquina } MP \text{ tal que } L = L(MP)\}$$

De esta definición se tiene que la clase de todos los problemas que tienen algoritmos probabilísticos tiempo polinomial con error de probabilidad igual a una constante menor que $\frac{1}{2}$, es llamada la clase BPP, sigla que viene de: error acotado tiempo polinomial (bounded, error probabilistic polinomial times)

Definición 3.23. *La clase R es el conjunto de todos los lenguajes que son decididos por una R-máquina, esto es.*

$$R = \{L : L \text{ es un lenguaje y existe una } R\text{-máquina } MP \text{ tal que } L = L(MP)\}$$

Definición 3.24. *La clase ZPP es el conjunto de todos los lenguajes que son decididos por una ZPP-máquina, es decir.*

$$ZPP = \{L : L \text{ es un lenguaje y existe una } ZPP\text{-máquina } MP \text{ tal que } L = L(MP)\}$$

Como en las BPP-máquinas, R-máquinas y las ZPP-máquinas, el error de probabilidad puede hacerse arbitrariamente pequeño en tiempo polinomial, se dice que los algoritmos probabilísticos relacionados con tales máquinas son algoritmos aleatorios eficientes, es por esto que las clases de complejidad probabilística definidas por tales máquinas son nociones de computación aleatoria eficiente. De esta manera se podría decir que estas clases de complejidad son "semejantes"a la clase P. Por otro lado, se tiene que en las máquinas PP el error de probabilidad no se puede hacer arbitrariamente pequeño en tiempo polinomial, entonces se puede decir que la clase de complejidad definida por las PP-máquinas es "semejante "a la clase NP.

¿Es $BPP = P$?. En este caso la aleatoriedad no aportaría nada nuevo a las ciencias de la computación. O quizá ¿$NP \subseteq BPP$?, es decir, ¿para cuales casos los problemas NP-completos tienen algoritmos aleatorios eficientes?. Desafortunadamente, las respuestas a las preguntas anteriores no se conocen. En este trabajo se mostrarán algunas propiedades algebraicas de las clases de complejidad probabilísticas y las relaciones entre las diferentes clases de complejidad.

Definición 3.25. *Sea C una clase de complejidad, se define la clase de complejidad coC como el conjunto $coC = \{L : L^c \in C\}$.*

Teorema 3.2.1. *[PC94] Las clases PP, BPP y ZPP son cerradas sobre el complemento. Además, las clases BPP, R, y ZPP son cerradas sobre la unión y la intersección.*

Demostración. La demostración será hecha por partes.

1. Se probará que las clases PP, BPP, y ZPP son cerradas sobre el complemento.

   Para la clase PP se debe probar que si un lenguaje $L \in PP$, entonces $L^c \in PP$.

   Sea L un lenguaje en PP, luego existe una PP-máquina MP que decide L.

   Sea MP' una PP-máquina la cual sobre la entrada x ejecuta MP y acepta si, y sólo si, MP rechaza. Se probará que la máquina MP' es una PP-máquina que decide L^c.

   Si $x \in L$, entonces $\alpha(MP, x) > \frac{1}{2}$ y de ahí, $\beta(MP', x) > \frac{1}{2}$, luego $MP'(x)$ rechaza la entrada.

   Si $x \notin L$, se tiene que $\beta(MP, x) > \frac{1}{2}$ y así, $\alpha(MP', x) > \frac{1}{2}$, luego $MP'(x)$ acepta la entrada.

   Para cualquier entrada x, se tiene que $MP'(x)$, acepta la entrada si $\alpha(MP', x) > \frac{1}{2}$ y rechaza la entrada si $\beta(MP', x) > \frac{1}{2}$. Es decir, existe una PP-máquina que decide L^c. Así, $L^c \in PP$. De manera similar se procede para demostrar que las clase BPP y ZPP son cerradas bajo el complemento.

2. Las clases BPP, R y ZPP son cerradas sobre la unión.

   Para la clase BPP se debe probar que, dados dos lenguajes L_1 y $L_2 \in PP$, existe una PP-máquina que decide $L_1 \cup L_2$, es decir $L_1 \cup L_2 \in PP$. Sean L_1 y L_2 lenguajes en BPP. Del teorema 3.1.1 se sigue que para cualquier constante $\epsilon \in (0, \frac{1}{2})$ existen máquinas MP_1 y MP_2 tales que, para $i = 1, 2$, si $x \in L_i$, entonces $\alpha(MP_i, x) > 1 - \epsilon$ y si $x \notin L_i$, entonces $\beta(MP_i, x) > 1 - \epsilon$.

Sea MP una BPP-máquina, la cual, sobre la entrada x, ejecuta sucesivamente $MP_1(x)$ y $MP_2(x)$ y acepta si, y sólo si, alguna de las dos ejecuciones acepta. La BPP-máquina MP se describe a continuación.

inicio{entrada: x}

ejecuta $MP_1(x)$

si $MP_1(x)$ acepta **entonces**

acepta

ejecuta $MP_2(x)$

si $MP_2(x)$ acepta **entonces**

acepta

rechaza

fin.

Si $x \in L_1 \cup L_2$, entonces $x \in L_1$ o $x \in L_2$.

Si $x \in L_1$, $MP_1(x)$ acepta siempre que $\alpha(MP_1, x) > 1 - \epsilon$. De igual manera, si $x \in L_2$, $MP_2(x)$ acepta siempre que $\alpha(MP_2, x) > 1 - \epsilon$. De lo anterior, se tiene que $MP(x)$ acepta si $\alpha(MP, x) > 1 - \epsilon$.

Si $x \notin L_1 \cup L_2$, entonces $x \notin L_1$ y $x \notin L_2$.

Si $x \notin L_1$, $MP_1(x)$ rechaza siempre que $\beta(MP_1, x) > 1 - \epsilon$. De igual manera, si $x \notin L_2$, $\beta(MP_2, x) > 1 - \epsilon$. Entonces, $\beta(MP, x) = \beta(MP_1, x)\beta(MP_2, x) > (1 - \epsilon)^2$

De lo anterior, se tiene que para cualquier entrada x.

$MP(x)$ acepta si $\alpha(MP, x) > 1 - \epsilon$ y rechaza si $\beta(MP, x) > (1 - \epsilon)^2$.

Escogiendo ϵ tal que $(1 - \epsilon)^2 > \frac{1}{2}$, para la constante $\epsilon' = (1 - \epsilon)^2 - \frac{1}{2} \in (0, \frac{1}{2})$ se tiene que:

$MP(x)$ acepta si $\alpha(MP, x) > (1 - \epsilon) > (1 - \epsilon)^2$. Luego $\alpha(MP, x) > (1 - \epsilon)^2 = \frac{1}{2} + (1 - \epsilon)^2 - \frac{1}{2} = \frac{1}{2} + \epsilon'$.

Además, $MP(x)$ rechaza si $\beta(MP, x) > (1 - \epsilon)^2 = \frac{1}{2} + \epsilon'$.

Por lo tanto, la BPP-máquina MP decide $L_1 \cup L_2$, así $L_1 \cup L_2 \in BPP$. De igual forma se prueba que las clases R y ZPP son cerradas sobre la unión.

3. Las clases BPP, R y ZPP son cerradas sobre la intersección.

   - Para la clase BPP.

     Sean L_1 y L_2 lenguajes en BPP, luego L_1^c y L_2^c también están en BPP, por (1). Además, $L_1^c \cup L_2^c \in BPP$ por (2), pero utilizando las leyes Demorgan se tiene que:

     $$L_1^c \cup L_2^c = (L_1 \cap L_2)^c \in BPP$$

     Por lo tanto, $((L_1 \cap L_2)^c)^c = L_1 \cap L_2 \in BPP$, esto es BPP es cerrada bajo la intersección. De igual manera, se prueba que la clase ZPP es cerrada bajo la intersección, ya que esta clase también es cerrada bajo el complemento y la unión.

   - Para la clase R

     Sean L_1 y L_2 lenguajes en R. Del teorema 3.1.2 y del teorema 3.1.1 se sigue que para cualquier constante $\epsilon \in (0, \frac{1}{2})$ existen máquinas MP_1 y MP_2 tales que para $i = 1, 2$ si $x \in L_i$ entonces $\alpha(MP_i, x) > 1 - \epsilon$ y si $x \notin L_i$, entonces $\beta(MP_i, x) = 1$.

     Sea MP una R-máquina, la cual, sobre la entrada x, ejecuta sucesivamente $MP_1(x)$ y $MP_2(x)$ y acepta si, y sólo si, $MP_1(x)$ y $MP_2(x)$ aceptan.

     $$\begin{aligned} & x \in L_1 \cap L_2 \Rightarrow \alpha(MP_i, x) > 1 - \epsilon \quad \text{para } i = 1, 2 \\ & \Rightarrow \alpha(MP, x) > (1 - \epsilon)^2 \end{aligned}$$

     Tomando ϵ tal que $(1 - \epsilon)^2 > \frac{1}{2}$, entonces $MP(x)$ acepta si $\alpha(MP, x) > \frac{1}{2}$

     Ahora,

     $$\begin{aligned} & x \notin L_1 \cap L_2 \Rightarrow \beta(MP_1, x) = 1 \quad \text{y} \quad \beta(MP_2, x) = 1 \\ & \Rightarrow \beta(MP, x) = 1. \end{aligned}$$

     Por lo tanto, hay una R-máquina que decide $L_1 \cap L_2$, es decir $L_1 \cap L_2 \in R$

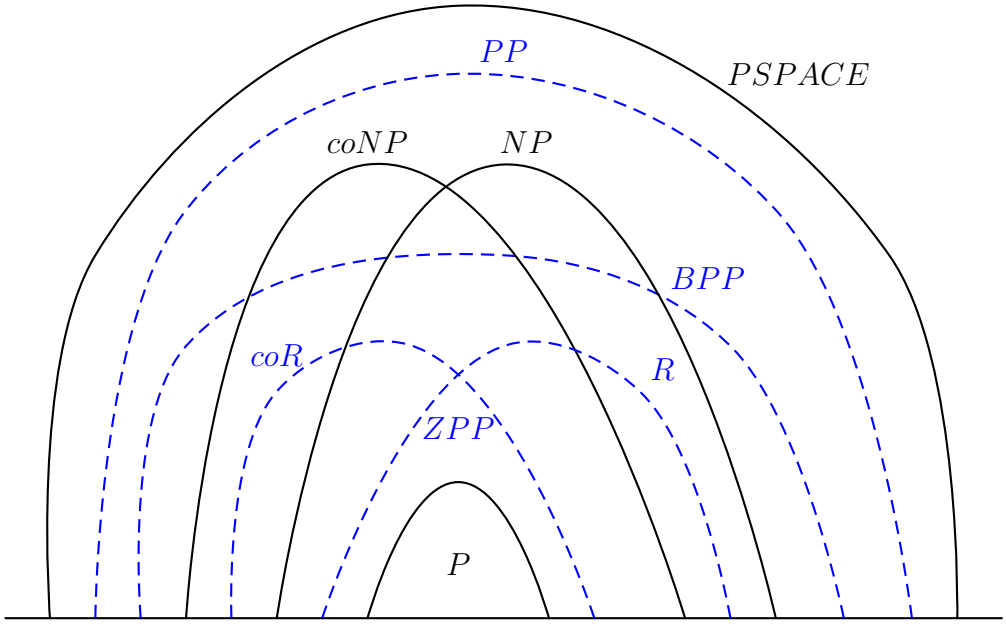


Figura 3.12: Relaciones entre las diferentes clases de complejidad

♢

En la figura 3.12 se muestran todas las inclusiones conocidas entre las diferentes clases de complejidad, las cuales se probarán a continuación. La clase de complejidad BPP y las contenidas en ella podrían considerarse como computacionalmente tratables, porque para estas clases el error de probabilidad puede hacerse arbitrariamente pequeño por un número polinomial de iteraciones. Para la clase P el error de probabilidad es cero.

Para la demostración del siguiente resultado es necesario la definición de la clase $PSPACE$.

Definición 3.26 (Clase $PSPACE$). *Un lenguaje L pertenece a la clase $PSPACE$ si existe un algoritmo determinístico A que lo decide y un polinomio $p(n)$, tal que la cantidad de almacenamiento requerido por A sobre cualquier entrada x de L no es más que $p(n)$, donde n es el tamaño de x.*

Teorema 3.2.2. *[PC94, Pap94]*

1. $P \subseteq ZPP$
2. $ZPP = R \cap coR$

3. $R \cup coR \subseteq BPP$

4. $R \subseteq NP \ y \ coR \subseteq coNP$

5. $BPP \subseteq PP$

6. $NP \cup coNP \subseteq PP$

7. $PP \subseteq PSPACE$

Demostración. Se probará cada una de las afirmaciones anteriores.

1. Sea $L \in P$, luego existe una máquina de Turing M determinística tiempo polinomial que decide L. Sea MP una ZPP-máquina tal que, para cualquier entrada x, los caminos de computación de $MP(x)$ son iguales a las computaciones de $M(x)$. Entonces se tiene que si $x \in L$, $\alpha(MP, x) = 1$ y si $x \notin L$, $\beta(MP, x) = 1$.

   Así, para cualquier $x \in L$, $\alpha(MP, x) = 1 > \frac{1}{2}$ y $\beta(MP, x) = 0$.

   Si $x \notin L$, $\beta(MP, x) = 1 > \frac{1}{2}$ y $\alpha(MP, x) = 0$. Esto es para $L \in P$, existe una ZPP-máquina MP que decide L, es decir $L \in ZPP$ y esto es $P \subseteq ZPP$.

2. Sea $L \in ZPP$, entonces existe una ZPP-máquina MP que decide L. por teorema 3.1.3 existe una máquina Turing probabilística MP' R-máquina que decide L, así $L \in R$, de donde $ZPP \subseteq R$.

   Como ZPP es cerrada bajo el complemento, si $L \in ZPP$, entonces

   $$L^c \in ZPP, \quad \text{pero, } ZPP \subseteq R \quad \Rightarrow L^c \in R \Rightarrow L \in coR$$

   Así $ZPP \subseteq coR$. Como $ZPP \subseteq R$ y $ZPP \subseteq coR$, entonces $ZPP \subseteq R \cap coR$.

   Para la otra contenencia, sea $L \in R \cap coR$, luego existen dos R-máquinas MP_1 y MP_2 que decide L y L^c, respectivamente. Sea MP la ZPP-máquina que se muestra a continuación.

   inicio{entrada: x}

   ejecuta $MP_1(x)$

si $MP_1(x)$ acepta **entonces**

acepta

ejecuta $MP_2(x)$

si $MP_2(x)$ acepta **entonces**

acepta

si no "no sé"

fin.

Como $L \in R \cap coR$ entonces $L \in R$ y $L \in coR$.

Si $x \in L$, entonces existen dos R-máquinas MP_1 y MP_2 tales que $\alpha(MP_1, x) > \frac{1}{2}$ y $\alpha(MP_2, x) > \frac{1}{2}$. Así, $\alpha(MP, x) > \frac{1}{2}$ y no hay computaciones de rechazo, esto es $\alpha(MP, x) > \frac{1}{2}$ y $\beta(MP, x) = 0$.

Ahora, si $x \notin L$, $\beta(MP_1, x) = 1$ y $\beta(MP_2, x) = 1$ así, $\beta(MP, x) = 1 > \frac{1}{2}$ y no hay computaciones de aceptación; luego $L \in ZPP$ y puesto que L es un lenguaje arbitrario en $R \cap coR$ se sigue que $R \cap coR \subseteq ZPP$.

Por lo tanto, $R \cap coR = ZPP$.

3. Sea $L \in R \cup coR$, entonces $L \in R$ o $L \in coR$.

   Si $L \in R$, existe una R-máquina MP_1 que decide L y por el teorema 3.1.2 existe una BPP-máquina MP_1' que decide L así, $L \in BPP$ de donde, $R \subseteq BPP$.

   Si $L \in coR$, existe una R-máquina MP_2 que decide L^c, por el teorema 3.1.2 existe una BPP-máquina MP_2' que decide L^c. Luego $L^c \in BPP$, pero la clase BPP es cerrada bajo el complemento lo cual se probó en el teorema anterior. Así, $(L^c)^c = L \in BPP$ entonces $coR \subseteq BPP$.

   Por lo tanto, $R \cup coR \subseteq BPP$.

4. *a*) Se Probará que $R \subseteq NP$

Sea $L \in R$, entonces existe una R-máquina MP que decide L. Sea M una máquina de Turing no determinística tal que para cualquier entrada x, $M(x)$ acepta si, y sólo si, $MP(x)$ acepta.

Si $x \in L$, entonces $\alpha(MP, x) > \frac{1}{2}$ y en $M(x)$ existe un camino de computación de aceptación y así $M(x)$ acepta. En caso contrario, es decir si $x \notin L$, se tiene que $\beta(MP, x) = 1$ y así ningún camino de computación de $M(x)$ acepta. De lo anterior, se tiene que existe una máquina de Turing no deterministica M que decide L, esto es $L \in NP$. Por lo tanto $R \subseteq NP$.

b) Sea $L \in coR$.

$$L \in coR \Leftrightarrow L^c \in R \Rightarrow L^c \in NP \quad \text{porque } R \subseteq NP$$
$$\Leftrightarrow L \in coNP$$

Así, $coR \subseteq coNP$

5. Sea $L \in BPP$, luego existe una BPP-máquina MP y algún $\epsilon \in (0, \frac{1}{2})$ tal que MP decide L de la siguiente manera.

   Si $x \in L$, entonces $\alpha(MP, x) > \frac{1}{2} + \epsilon$ y si $x \notin L$, $\beta(MP, x) > \frac{1}{2} + \epsilon$. Pero $\frac{1}{2} + \epsilon > \frac{1}{2}$. Entonces, $\alpha(MP, x) > \frac{1}{2}$ siempre que $x \in L$ y $\beta(MP, x) > \frac{1}{2}$ siempre que $x \notin L$.

   Esto es, si $x \in L$, entonces $\alpha(MP, x) > \frac{1}{2}$ y si $x \notin L$, $\beta(MP, x) > \frac{1}{2}$. Es decir, la misma máquina MP interpretada como una PP-máquina decide L.

   Por lo tanto, existe una PP-máquina que decide L, es decir $L \in PP$.

   Así, $BPP \subseteq PP$

6. Sea $L \in NP \cup coNP$, entonces $L \in NP$ o $L \in coNP$

   a) Si $L \in NP$, entonces existe una máquina de Turing no determinística M tiempo polinomial tal que $L = L(M)$.

   Sea MP una PP-máquina, la cual, sobre la cadena x, aleatoriamente, escoge para ejecutar uno de los siguiente pasos:

   1) Ejecuta $M(x)$.

2) Ejecuta $M(x)$ y no tiene en cuenta las computaciones obtenidas, siempre acepta, es decir los caminos de computación siempre terminan en computaciones de aceptación

y acepta si, y sólo si, $MP(x)$ acepta.

Considere una cadena x. Suponga que M sobre x computa $p(|x|)$ pasos y produce $2^{p(|x|)}$ computaciones. ¿Cuántas computaciones produce la máquina MP?, como MP es idéntica a M, pero MP puede escoger entre dos opciones la instrucción a ejecutar y además, para $p(|x|)$ pasos M produce $2^{p(|x|)}$ computaciones, entonces MP, para $p(|x|)$ pasos produce $2 * 2^{p(|x|)} = 2^{p(|x|)+1}$ computaciones. De estas al menos $\frac{1}{2}$ paran en "sí"(las que corresponde a la mitad de las computaciones que aceptan incondicionalmente). Así, la mayoría de la computaciones de MP aceptan si, y sólo si, al menos una computación de M sobre x, acepta.

Por lo tanto, MP decide L si $\alpha(MP, x) > \frac{1}{2}$, luego existe una PP-máquina MP que decide L, esto es $L \in PP$ y así $NP \subseteq PP$

b) Por la definición de $coNP$ se tiene que, $L \in coNP$ si, y sólo si, $L^c \in NP$.

De $6a$ se tiene que $NP \subseteq PP$, entonces $L^c \in PP$. Lo cual implica que, $(L^c)^c = L \in PP$ porque PP es cerrada bajo el complemento.

Así, $coNP \subseteq PP$.

Por lo tanto, por $6a$ y $6b$ se tiene que $(NP \cup coNP) \subseteq PP$

7. Sea $L \in PP$, luego existe una PP-máquina MP que decide L en tiempo $p(n)$ donde p es un polinomio y $n = |x|$. Entonces L puede ser decidido por el siguiente algoritmo determinístico.

inicio{Entrada: x}

$n \leftarrow |x|$

$cca \leftarrow 0$\{número de caminos de computaciones de aceptación\}

para $i \leftarrow 1$ **hasta** $2^{p(n)}$ **hacer**

si el i-ésimo camino de computación de $MP(x)$ acepta **entonces**

$cca \leftarrow cca + 1$ {verifica si $\alpha(MP, x) > \frac{1}{2}$}

si $cca > 2^{p(n)-1}$ **entonces**

acepta

si no

rechaza

fin.

Puesto que la ejecución de un camino de computación de $MP(x)$ requiere a lo más un número polinomial de celdas[31]y puesto que el $i+1$-ésimo camino de computación puede ser ejecutado utilizando la misma celda que se usó en la i-ésima computación, porque todas las computaciones de una máquina de Turing probabilística sobre una entrada x tienen el mismo número de pasos, entonces, el algoritmo utiliza un número polinomial de celdas para la ejecución de las computaciones, así $L \in PSPACE$, esto es, $PP \subseteq PSPACE$.

$\diamondsuit$

3.3. LENGUAJES PP-COMPLETOS

Para iniciar el estudio de los lenguajes PP-completos se deben definir las reducciones polinomiales y los lenguajes A-completos, donde A es un clase de complejidad arbitraria.

Definición 3.27 (Reducción polinomial). *Sean Σ_1^* y Σ_2^* alfabetos. Una reducción polinomial de un lenguaje $L_1 \in \Sigma_1^*$ a un lenguaje $L_2 \in \Sigma_2^*$ es una función $f : \Sigma_1^* \to \Sigma_2^*$ que satisface las siguientes condiciones:*

1. *Existe una máquina de Turing tiempo polinomial que computa f.*
2. *Para toda $x \in \Sigma_1^*$, $x \in L_1$ si, y sólo si, $f(x) \in L_2$.*

En este caso se escribe $L_1 \leq_P L_2$ y se lee "L_1 se reduce polinomialmente a L_2"

[31]De la primera parte de este capítulo, recuerde que la máquina de Turing utiliza para la entrada y salida de datos una cinta bi-infinita dividida en celdas.

La reducción polinomial cumple la ley transitiva, como se mostrará en el siguiente teorema, el cual está demostrado en [HB03].

Teorema 3.3.1. *Sean L_1, L_2 y L_3 lenguajes. Si $L_1 \leq_p L_2$ y $L_2 \leq_p L_3$ entonces $L_1 \leq_p L_3$.*

Definición 3.28 (Lenguaje completo). *Un lenguaje L es A-completo si:*

1. $L \in A$.
2. $L' \leq_p L$ *para cada* $L' \in A$.

Por ejemplo, en la clase NP se tiene que el problema de Satisfabilidad (SAT) es NP-completo [CLR90].

No se sabe si los lenguajes completos existen para las clases BPP, R, y ZPP. Hay algunas evidencias de que no existen tales lenguajes [PC94]. En esta sección se probará que los lenguajes PP-completos si existen. Así, como los problemas NP-completos son considerados como los más duros en NP, los problemas PP-completos se considerán como los más duros en PP.
Primero se probará que la clase PP es cerrada bajo la reducción polinomial.

Teorema 3.3.2. *Sean L_1 y L_2 lenguajes tales que $L_1 \leq_p L_2$. Si $L_2 \in PP$, entonces $L_1 \in PP$.*

Demostración. Sean Σ_1^* y Σ_2^* los alfabetos de L_1 y L_2 respectivamente. Por hipótesis $L_1 \leq_p L_2$, es decir existe una reducción polinomial $f : \Sigma_1^* \to \Sigma_2^*$ de L_1 en L_2.

Sea MTf la máquina de Turing tiempo polinomial que computa f. Como $L_2 \in PP$, existe una PP-máquina MP_2 que decide L_2, se debe probar que $L_1 \in PP$, esto es, que existe una PP-máquina MP_1 que decide L_1.

La MP_1 puede ser construida de la siguiente forma: para cada entrada $x \in \Sigma_1^*$, MP_1 utiliza MTf para transformar x en $f(x) \in \Sigma_2^*$ y luego usa MP_2 para determinar si $f(x) \in L_2$.

- Si $f(x) \in L_2$, entonces $\alpha(MP_2, f(x)) > \frac{1}{2}$ y $MP_1(x)$ acepta. Así, $\alpha(MP_1, x) > \frac{1}{2}$.
- Si $f(x) \notin L_2$, entonces $\beta(MP_2, f(x)) > \frac{1}{2}$ y $MP_1(x)$ rechaza. Así, $\beta(MP_1, x) > \frac{1}{2}$.

Puesto que $x \in L_1$ si, y sólo si, $f(x) \in L_2$, se tiene que MP_1 decide L_1. Además MP_1 es una PP-máquina, luego $L_1 \in PP$. $\diamondsuit$

De manera análoga, se prueba que las clases BPP, R, y ZPP son cerradas bajo reducciónes polinomiales.
Considere el siguiente problema.

Máxima satisfablidad

Entrada: Una fórmula booleana f sobre un conjunto U finito de variables y un entero no negativo i .

Pregunta: ¿Es f satisfaciente por más de i de las posibles asignaciones de verdad de sus variables?

LEMA 5. *[PC94] máxima satisfabilidad* $\leq_p$ $\frac{1}{2}$*-satisfabilidad.*

Demostración. Sea $\langle f, i\rangle$, una entrada de máxima satisfabilidad donde f es una fórmula en FNC sobre el conjunto $U = \{x_1, x_2, \ldots, x_n\}$ de variables e i un número entero más pequeño que 2^n. Sea $i = 2^{n-r_1} + 2^{n-r_2} + \cdots + 2^{n-r_k}$ con $1 \leq r_1 < r_2 < \ldots < r_k \leq n$. Ahora, se define g_i, una fórmula en forma normal disyuntiva(FND), de la siguiente manera:

$$
\begin{aligned}
g_i = & (x_1 \wedge x_2 \wedge \cdots \wedge x_{r1}) \\
& \vee (\sim x_1 \wedge \sim x_2 \wedge \ldots \wedge \sim x_{r_1} \wedge x_{r_1+1} \wedge \cdots \wedge x_{r_2}) \\
& \vee (\sim x_1 \wedge \sim x_2 \wedge \ldots \wedge \sim x_{r_2} \wedge x_{r_2+1} \wedge \cdots \wedge x_{r_3}) \\
& \cdots \\
& \cdots \\
& \vee (\sim x_1 \wedge \sim x_2 \wedge \ldots \wedge \sim x_{r_{k-1}} \wedge x_{r_{k-1}+1} \wedge \cdots \wedge x_{r_k})
\end{aligned}
$$

Para la primera cláusula de g_i se tienen 2^{n-r_1} asignaciones de verdad que la satisfacen. En efecto, para que la cláusula sea satisfaciente la asignación de verdad debe ser $t(x_i) = v$, para $i = 1, 2, \ldots, r_1$. Las variables restantes pueden tomar cualquier valor de verdad, luego hay 2^{n-r_1} posibilidades de tomar los valores de verdad para las variables $(x_{r_1+1}, x_{r_1+2}, \ldots, x_k)$, así, el número de asignaciones de verdad satisfacientes para la primera cláusula es 2^{n-r_1}. De igual manera, se tiene que el número de asignaciones de verdad satisfacientes para la h-ésima cláusula de g_i es 2^{n-r_h} con $1 \leq h \leq k$, recuerde que, la h-ésima cláusula de g_i es de la forma $(\sim x_1 \wedge \sim x_2 \wedge \cdots \wedge \sim x_{r_{h-1}} \wedge x_{r_{h-1}+1} \wedge \ldots \wedge x_{r_h})$, luego, la asignación de verdad para que esta cláusula sea satisfaciente es $t(x_i) = v$, para $i = r_{r_{h-1}+1}, r_{r_{h-1}+2}, \ldots, r_{r_h}$, para el resto variables $x_{r_h+1}, x_{r_h+2}, \ldots, x_{r_k}$ hay 2^{n-r_h} posibilidades de tomar cualquier valor de verdad y así se tiene que hay 2^{n-r_h} asignaciones de verdad que satisfacen la h-ésima cláusula de g_i. Además, las variables negadas aseguran que tales asignaciones de verdad satisfacen sólo una cláusula, por ejemplo, para la primera cláusula hay 2^{n-r_1} asignaciones de verdad satisfacientes y estas asignaciones no satisfacen otra cláusula, ya que, en las cláusulas siguientes las variables $x_1, x_2, \ldots, x_{r_1}$ están negadas. Por lo tanto, el número total de asignaciones de verdad que satisfacen g_i es exactamente $2^{n-r_1} + 2^{n-r_2} + \cdots + 2^{n-r_k} = i$. Mientras que el número de asignaciones de verdad que satisfacen $\sim g_i$ es precisamente $2^n - i$.

Sea g una fórmula en FND sobre el conjunto $U' = \{x_1, x_2, \ldots, x_n, y\}$ de variables definida de la siguiente manera:

$$g = (y \wedge f) \vee (\sim y \wedge \sim g_i)$$

Si f es satisfaciente por al menos i asignaciones de verdad, entonces g es satisfaciente por más de la mitad de las asignaciones de verdad. En efecto, f es satisfaciente para al menos i asignaciones de verdad y $\sim g_i$ es satisfaciente para exactamente $2^n - i$ asignaciones de verdad. Luego g es satisfaciente para al menos 2^n asignaciones de verdad. Como g es una fórmula sobre U' y $|U'| = n + 1$, entonces el número de posibles asignaciones de verdad para las variables en U' es 2^{n+1}. Por lo tanto, g es satisfaciente por más de la mitad de las asignaciones de verdad para las variables en U'.

Por otro lado, g es satisfaciente por más de la mitad de las asignaciones de verdad, es decir, g es satisfaciente por más de 2^n asignaciones de verdad y $\sim g_i$ es satisfaciente por precisamente $2^n - i$. Entonces f es satisfaciente por al menos i asignaciones de verdad.

Por lo tanto, f es satisfaciente para al menos i asignaciones de verdad si, y sólo si, g es satisfaciente por más de la mitad de las posibles asignaciones de verdad para sus variables.

Para construir g_i se necesitan a lo más n^2 pasos, ya que g_i podría tener a lo más n cláusulas y cada cláusula puede tener a lo más n literales; de igual manera, para construir $\sim g_i$ se necesitan a lo más n^2 pasos. De ahí, g puede ser construida a lo más en $2n^2$ pasos. Esto es, existe una MTg que computa g en tiempo polinomial. Por lo tanto, máxima satisfabilidad se reduce polinomialmente a $\frac{1}{2}$-satisfabilidad. $\diamondsuit$

Mostrar que un problema es PP-completo, sería mostrar que todo problema en PP se reduce polinomialmente a él, lo cual no es una tarea fácil, más aún, no se conocen cuantos problemas PP-completos existen en realidad. El siguiente teorema proporciona una herramienta más práctica para probar PP-completitud, si se tiene un problema que ya es PP-completo.

Teorema 3.3.3. *Sean L_1 y L_2 lenguajes. Si L_1 y L_2 están en PP, L_1 es PP-completo y si $L_1 \leq_p L_2$ entonces L_2 es PP-completo.*

Demostración. Como $L_2 \in PP$, sólo falta probar que, para todo $L' \in PP$, L' se reduce polinomialmente a L_1, es decir $L' \leq_p L_1$.
Sea L' cualquier lenguaje en PP. Como L_1 es PP-completo, se debe tener que
$L' \leq_p L_1$. Además, $L_1 \leq_p L_2$ por hipótesis, entonces $L_1 \leq_p L_2$ por la transitividad de la reducción polinomial vista en el teorema 3.3.1.
Por lo tanto, L_2 es PP-completo $\diamondsuit$

Teorema 3.3.4. *[PC94] máxima satisfabilidad es PP-completo.*

Demostración. Para probar que *máxima satisfabilidad* es PP-completo se debe tener:

1. *máxima satisfabilidad* $\in PP$

   Por el lema 5, se tiene que *máxima satisfabilidad* se reduce polinomialmente a $\frac{1}{2}$-satisfabilidad, pero $\frac{1}{2}$-satisfabilidad $\in PP$ y PP es cerrada bajo reducciones polinomiales, entonces *máxima satisfabilidad* $\in PP$.

2. $L \leq_p$ *máxima satisfabilidad*, para cualquier $L \in PP$

   Recuerde de la demostración del teorema de Cook[32] [PC94, HB03], la idea que cualquier lenguaje $L \in NP$ se reduce polinomialmente a SAT. La cual es: para un lenguaje $L \in NP$ se tiene que existe una máquina de Turing no determinística tiempo polinomial que decide L, luego si $x \in L$, en el árbol de computación asociado a $M(x)$, existe una camino de computación de aceptación. Así, $f_L(x)$ (reducción polinomial) construye una colección de cláusulas sobre un conjunto U de variables, tal que, una asignación de verdad es una asignación de verdad satisfaciente si, y sólo si, esta es la asignación de verdad inducida por la computación de aceptación para $x \in L$, en otras palabras, $f_L(x)$ "codifica" las computaciones de $M(x)$.

   Se probará que para cualquier $L \in PP$, $L \leq_p$*máxima satisfabilidad*. Sea $L \in PP$, luego existe una PP-máquina MP la cual decide L en tiempo polinomial, sea $p(n)$, donde $n = |x|$, el polinomio que acota el tiempo de ejecución de MP. Sea f_x la fórmula booleana que "codifica" las computaciones de $MP(x)$ de igual manera que en la demostración del teorema de Cook.

   Como el tiempo de ejecución de MP está acotado por $P(n)$, entonces el árbol de computación asociado a $MP(x)$ tiene a lo más $2^{p(n)}$ caminos de computación. Si $x \in L$, $\alpha(MP, x) > \frac{1}{2}$, luego el número de computaciones de aceptación en el árbol de computación es mayor que $\frac{2^{p(n)}}{2}$. Por la construcción de f_x, una asignación de verdad es satisfactible si, y sólo si, ésta es la asignación de verdad inducida por un camino de computación de aceptación, esto es, por cada computación de aceptación se tiene una asignación de verdad satisfaciente para la fórmula booleana f_x.

[32] El cual prueba que SAT es NP-completo

Finalmente, la reducción es definida como $h(x) = \langle f_x, 2^{p(n)-1} \rangle$. Si $x \in L$, $h(x)$ está en *máxima satisfabilidad* y si $h(x)$ está en *máxima satisfabilidad* en el árbol de computación asociado con $h(x)$ hay más de $2^{p(n)-1}$ caminos de computación que terminan en computaciones de aceptación, es decir más de la mitad de las computaciones de $MP(x)$ paran en computación de aceptación, es decir $x \in L$. Así para cualquier x, $x \in L$ si, y sólo si, $h(x) \in$*máxima satisfabilidad*.

Para cualquier x. $h(x)$ es computable en tiempo polinomial porque f_x se puede construir en tiempo polinomial[33] para cualquier entrada x y el número de asignaciones de verdad satisfacientes para f_x también son construidas en tiempo polinomial, ya que, estas se obtienen del árbol de computación asociado a $MP(x)$ y MP es polinomial.

Por lo tanto, $h(x)$ se computa en tiempo polinomial y $x \in L$ si, y sólo si, $h(x)$ está en *máxima satisfabilidad*.

Así, $L \leq_p$*máxima satisfabilidad* para cualquier $L \in PP$.

Por lo tanto, *máxima satisfabilidad* es PP-completo. $\diamondsuit$

Corolario 3.4. *[PC94]* $\frac{1}{2}$*-satisfabilidad es* PP*-completo.*

Demostración. Por el teorema 3.3.4 *máxima satisfabilidad* es PP-completo, entonces por el teorema 3.3.3 $\frac{1}{2}$-satisfabilidad es PP-completo, ya que $\frac{1}{2}$-satisfabilidad$\leq_p$*máxima satisfabilidad*. $\diamondsuit$

[33]Esto se garantiza en la demostración del teorema de Cook.

4. CONCLUSIONES

- Se realizó un estudio detallado de los diferentes tipos de algoritmos probabilísticos que se conocen: Algoritmos de las Vegas y algoritmos de Monte Carlo.

- En el documento se presentó una aplicación de la teoría de la probabilidad en matemática computacional.

- Se mostró la aplicación de los algoritmos probabilísticos en problemas de lógica, en teoría de grafos, en polinomios multivariados, en teoría de números y en otros campos.

- Se hizo un estudio minucioso y detallado del modelo de computación Máquina de Turing probabilística y sus diferentes tipos de máquinas, con el fín de formalizar el concepto de algoritmo probabilístico y definir las clases de complejidad probabilística.

- En el estudio realizado de las diferentes clases de máquinas de Turing probabilísticas, se mostró que en las máquinas BPP, R y ZPP, el error de probabilidad se puede hacer arbitrariamente pequeño en tiempo polinomial, es decir, los algoritmos relacionados con estas máquinas son "eficientes"y devuelven respuestas correctas con una alta probabilidad.

- Se estudiaron e ilustraron las relaciones entre cada una de las clases de complejidad definidas por las maquinas de Turing probabilísticas.

- Las diferentes clases de complejidad determinadas por cada uno de los tipos de máquinas de Turing probabilísticas fueron estudiadas detalladamente. Los resultados mostrados para las diferentes clases de complejidad probabilística se demostraron con los detalles y justificaciones pertinentes de tal manera que sean accesibles a toda persona interesada en el tema con un nivel básico de preparación en matemática computacional.

- Al estudiar por primera vez la teoría de los algoritmos probabilísticos se encuentra la dificultad de que la literatura disponible para este tipo de algoritmos es muy escasa y de un nivel avanzado. En el seminario de grado se hizo un presentación amplia y detallada de este tipo de algoritmos.

BIBLIOGRAFÍA

[PC94] PIERRE BOVE, Daniel; CRESSCENZi, Pierluig.Introdution to the theory of complexity. Gran Bretaña, Prentice-hall international. 1994

[Pap94] PAPADIMITRIOU, Christos H. Computational Complexity. Addison Wesley, publishing Company.1994.

[CLR90] CORMEN, Thomas; LEISERSON, Charles y RIVEST, Ronald. Introduction to Algorithms. Nueva york. McGraw Hill.1990.

[HEW86] HERBERT, Wilf. Algorithms and complexity. Nueva york. Prentice-hall international.1986.

[BB97] BRASSARD, G; BRATLEY, P. Fundamentos de algoritmia. Prentice-Hall. 1997.

[HB03] HERRERA FLOREZ, Maritza; BOLAÑOS RIVERA, Yudy Marcela. Aproximabilidad en problemas NP-duros. Universidad del Cauca, facultad de ciencias naturales exactas y de la educación. Popayán. 2003.

[FRA87] FRALEIGH, JOHN B. Álgebra abstracta. Addison–wesley iberoaméricana. 1987.

[KEN97] KENNERH H.ROSEN. Elemetary number theory and its applications. Nueva york. Addison-wesley publishing company.1997.

[JGR99] JIMÉNEZ Rafael, GORDILLO Enrique, RUBIANO Gustavo. Teoría de numeros para principiantes. Bogotá. Universidad Nacional de Colombia facultad de ciencias. 1999.

[Rab77] RABIN Michael O. Probabilistic algorithm for testing primality. Institute of mathematics, Hebrew university. Jerusalem, Israel. 1977

[GJ79] GAREY, Michael R y JOHNSON, David S. Computers and intractibility: A guide to the theory of NP- completeness. New york: W.H. Freeman, 1979.

[Pa71] PARZEN, Emanuel. Teoría Moderna de Probabilidades y sus aplicaciones. Editorial Limusa. 1971.

[De88] DEGROOT, M. Probabilidad y Estadística. México. Addison Wesley Iberoamericana. Segunda Edición. 1988.

[Ko88] KOROLIUK, V. S. Manual de Teoría de Probabilidades y Estadística Matemática Editorial Mir. Primera Reimpresión 1981. Segunda Edición. 1998.

[Cas] CASTRO d. Algoritmos probabilísticos. Disponible en Internet: www.decsai.urg.es/ castro/CA/node12. html.

[Mor1] MORALES g. Computabilidad y complejidad. Disponible en Internet: www.delta.cs.cinvestav.mx/ gmorales/complex / node1. Html.

[Mor2] MORALES g. Algoritmos probabilísticos y su jerarquía. Disponible en Internet www. delta.cs.cinvestav.mx/ gmorales/complex/node215.html

[FH02] FORTNOW Lance, HOMER Steve. A short history of computational complexity. university of Boston, 2002. <http:// www.researchindex.com>.

[Coo00] COOK, Stephen. the P versus NP Problem. University of Toronto, 2000.<http:// www.claymath.org/Millenium-Prize- Problems/P-vs- NP/-Objects/Official-Problem-Description.pdf>.