Algoritmos y Complejidad

Algoritmos Probabilísticos

Pablo R. Fillottrani

Depto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Primer Cuatrimestre 2017

Algoritmos Probabilísticos

Introducción

- un algoritmo probabilístico es un algoritmo que contiene entre sus operaciones elementales la generación de números aleatorios
- dado que no es un proceso determinístico, algunos autores no los consideran verdaderos “algoritmos”
- su principal característica es que el mismo algoritmo aplicado a la misma instancia varias veces puede dar distintos resultados, y por lo tanto necesitar distinta cantidad de tiempo y espacio.

- esto permite que alguna ejecución de un algoritmo probabilístico devuelva un resultado incorrecto, o cicle indefinidamente, siempre y cuando lo haga con una probabilidad baja
- en muchos casos se puede aumentar la confianza en el resultado ejecutando el algoritmo varias veces

- el análisis de los algoritmos probabilísticos es frecuentemente muy complejo, y requere herramientas de probabilidad y estadística fuera de los alcances de este curso
- hay que distinguir entre:
  - el tiempo promedio de un algoritmo determinístico es el promedio del tiempo que toma el algoritmo cuando cada instancia de un mismo tamaño es igualmente probable
  - el tiempo esperado de un algoritmo probabilístico se define para cada instancia particular: es el tiempo medio que llevaría resolver la instancia varias veces. No depende de ninguna distribución de probabilidades de las instancias; sí de la distribución de probabilidades de los números aleatorios generados
- tiene sentido entonces hablar del tiempo esperado promedio, y del tiempo esperado en el peor caso de un algoritmo probabilístico para un dado tamaño de instancias

Variable aleatoria de evento

- es útil para el análisis probabilístico de los algoritmos
- sea A un evento

$$X_A = \begin{cases} 1 & \text{si } A \text{ ocurre} \\ 0 & \text{si } A \text{ no ocurre} \end{cases}$$

- se puede probar que $E[X_A] = Pr\{A\}$
- también es útil $E[X+Y] = E[X] + E[Y]$

PROBLEMA DE LA BUSQUEDA LABORAL

- se quiere contratar una nueva secretaria mediante una agencia laboral, que nos envía un candidato por día. Entrevistarlo cuesta c_e. Si después de la entrevsta se decide que tiene mejores antecedentes que la secretaria actual, entonces el costo de despedirla y contratar a la nueva es de c_c, con $c_c >> c_e$

```
FUNCTION BUSQUEDALABORAL (n)
  actual ::= 0
  FOR i ::= 1 TO n
     entrevista candidato i
     IF candidato i es mejor que actual
         actual := i; contratar i
     ENDIF
  ENDFOR
  RETURN
```


PROBLEMA DE LA BUSQUEDA LABORAL

- se desea saber el costo de esta estrategia
- en el peor caso $\Theta(c_e * n + c_c * n) = \Theta(c_c * n)$
- para realizar un análisis probabilístico es necesario conocer la distribución de los datos de entrada, lo que es en general bastante difícil de establecer
- en este caso se puede asumir que los candidatos vienen en orden aleatorio, es decir que la lista de los órdenes de candidatos de acuerdo a sus antecedentes es es igualmente probable con cada una de las $n!$ permutaciones posibles
- esto es, los candidatos forman una permutación aleatoria uniforme

PROBLEMA DE LA BUSQUEDA LABORAL

- sea X_i el evento en que el candidato i es contratado, por el lema mencionado vale $E[X_i] = Pr\{i \text{ sea contratado}\}$ y esta probabildad es $1/i$ si todos los candidatos llegan en orden aleatorio
- entonces si X es el número total de contrataciones que se hacen, $X = \sum_{i=1}^{n} X_i$, y

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} PrX_i =$$

$$= \sum_{i=1}^{n} 1/i = ln(n) + O(1) \in O(\log(n))$$

- el costo total promedio del algoritmo es $O(c_c * \log(n))$

Generación de números aleatorios

- se supondrá disponible en los algoritmos probabilísticos una sentencia de generación de números aleatorios de costo constante:

  $$\texttt{uniforme}(0,1)$$

  que devuelve un número real x uniformemente distribuido en el intervalo $[0,1)$
- llamadas sucesivas generaran una secuencia de valores independientes
- en la práctica no es posible una implementación estricta de esta sentencia, por la propiedad de que los números generados sean realmente aleatorios. Existen, sin embargo, aproximaciones que alcanzan para la mayoría de las aplicaciones

- en base a esta sentencia, se pueden obtener otros tipos de valores aleatorios:

```
procedure uniforme(a,b)
  RETURN a+(b-a)*uniforme(0,1)

procedure uniforme(i...j)
  RETURN piso(uniforme(i,j+1))

procedure tirarMoneda
  IF uniforme(0..1)
    RETURN cara
  ELSE
    RETURN ceca
  ENDIF
```


- para proveer $\texttt{uniforme}(0,1)$ la mayoría de las veces los lenguajes de programación implementan generadores pseudoaleatorios que son procedimientos determinísticos que generan una larga secuencia de valores aparentemente aleatorios
- para comenzar la secuencia se provee un valor incial llamado semilla (la misma semilla da origen a la misma secuencia)
- se usan dos funciones $f : X \longrightarrow X$ y $g : X \longrightarrow Y$ donde X es el dominio de la semilla (debe ser un conjunto grande), e Y es el dominio de los valores pseudoaleatorios

$$\begin{cases} x_0 = s \\ y_i = g(x_{i-1}),\ x_i = f(x_{i-1}) \quad \text{si } i > 0 \end{cases}$$

- hay que tener cuidado con los generadores provistos por los lenguajes de programación porque muchas veces no están bien implementados, lo que invalida los resultados y la confianza del algoritmo probabilístico
- una función simple para ejemplificar la generación de valores booleanos pseudoaleatorios podría ser la siguiente:

$$f(x) = x^2 \textbf{ mod } n \qquad\qquad g(x) = x \textbf{ mod } 2$$

donde n es el producto de dos números primos grandes, y la semilla inicial se elige entre los números de $[0, n-1]$ que son relativamente primos con n.

- para la mayoría de las aplicaciones, y siempre para valores booleanos, generadores más rápidos pero menos seguros como

$$f(x) = ax + b \textbf{ mod } n \qquad\qquad g(x) = x \textbf{ mod } 2$$

para determinados valores de a, b, n, pueden ser apropiados.

Clasificación

- de acuerdo al resultado que obtienen, los algoritmos probabilísticos se clasifican en:
  - numéricos: dan como resultado un rango de confianza del valor pretendido. Ejemplo: "el valor buscado está en el intervalo $[x, y]$". Dos aplicaciones muy importantes son las encuestas y la simulación
  - Monte Carlo: dan como resultado el valor correcto con una probabilidad alta, pero pueden a veces dar valores equivocados. Se puede reducir la incertidumbre ejecutando varias veces el algoritmo
  - Las Vegas: nunca dan un resultado incorrecto, pero a veces pueden detectar que el resultado es imposible de obtener en esa ejecución y devuelven un mensaje de error

- para ilustrar esta clasificación, si se tuviera un algoritmo de cada clase que compute el año del descubrimiento de América los resultados en distintas llamadas podrían ser:
  - numérico: "entre 1490 y 1500 con 99 % de probabilidad", "entre 1485 y 1495 con 90 % de probabilidad", "entre 1480 y 1490 con 95 % de probabilidad".
  - Monte Carlo: 1492,1492,1492,1491,1492,32134,1492,1492
  - Las Vegas: 1492,1492,error,1492,1492,1492,error,1492

Integración Numérica

- Problema: calcular $I = \int_a^b f(x)dx$.
- puede resolverse con un algoritmo probabilístico, que determina n puntos aleatorios entre a y b, calcula el valor de la función y aproxima la integral con el promedio de estos valores
- es un algoritmo probabilístico numérico

Algoritmo

```
FUNCTION Integral (f,n,a,b)
  suma ::= 0
  FOR i ::= 1 TO n
     x ::= uniforme(a,b)
     suma ::= suma+f(x)
  ENDFOR
  RETURN (b-a)*(suma/n)
```

- la varianza del valor calculado es inversamente proporcional a $\sqrt{n}$, lo que implica que n tiene que aumentar 100 veces lo hecho hasta entonces si se quiere un dígito adicional de precisión

- existen algoritmos determinísticos mejores que aproximan más rápido que este algoritmo probabilístico; pero la diferencia está en que no existen funciones en las que el algoritmo probabilístico siempre de un mal resultado
- variantes de este algoritmo se usan en la práctica con integrales cuartas o de nivel mayor
- en estos casos la cantidad de puntos para la muestra no aumenta tanto con los algoritmos probabilísticos como con los determinísticos

Análisis

- el análisis de este algoritmo resultaría en

$$Pr[|X - E(X)| < \varepsilon] \leq c\%$$

lo que significa que en el $c\%$ de los casos el error absoluto es menor que ε
- los valores de c y ε dependen de n y de la varianza de la variable aleatoria

Verificación del Producto de Matrices

- los algoritmos Monte Carlo se aplican a problemas para los que no existen algoritmos eficientes que siempre devuelvan la solución correcta
- el hecho de que ocasionalmente den una respuesta equivocada no significa que en determinadas instancias la mayoría de las veces el algoritmo falla
- la probabilidad de falla debe ser baja para todas las instancias
- sea $0 < p < 1$, un algoritmo Monte Carlo se dice p-correcto si retorna una respuesta correcta con probabilidad al menos p en todas las instancias

- Problema: se tienen tres matrices A, B, C de $n \times n$, y se quiere saber si $C = AB$
- la manera obvia de verificarlo es calcular AB y comparar el resultado con C, toma tiempo $\Theta(n^3)$ (o $\Theta(n^{\log_2 7})$ con Strassen)

- un algoritmo probabilístico más eficiente consiste en testear o no cada fila de AB con C de acuerdo a una elección aleatoria
- si $AB = C$ entonces el resultado siempre será igual
- si $AB \neq C$ entonces existe al menos un elemento diferente
- la fila de ese elemento será testeada con probabilidad $0,5$ en cuyo caso se encontrará la diferencia. Por lo tanto es un algoritmo $0,5$-correcto
- es un algoritmo Monte Carlo

Algoritmo

```
FUNCTION Freivalds (A,B,C,n)
  FOR i ::= 1 TO n
     X[i]::= uniforme(0..1)
  ENDFOR
  IF  (XA)B=XC
     RETURN true
  ELSE
     RETURN false
  ENDIF
```

- el tiempo de ejecución de este algoritmo es de $O(n^2)$ (porqué?)

- ejemplo:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{pmatrix} \quad C = \begin{pmatrix} 11 & 29 & 37 \\ 29 & 65 & 91 \\ 47 & 99 & 45 \end{pmatrix}$$

- si $X = (1,1,0)$ entonces $(XA)B = (40,94,128)$, $XC = (40,94,128)$, y la respuesta es **true**
- si $X = (0,1,1)$ entonces $(XA)B = (76,166,236)$, $XC = (76,164,136)$ y la respuesta es **false**

- una probabilidad de error de 50 % no es buena; sería más eficiente decidir si son iguales o no en base a tirar una moneda
- el punto central es que cuando el algoritmo retorna **false** estamos seguro que la respuesta es correcta. Solamente cuando la respuesta es **true** se tienen dudas
- esta característica permite aumentar la confianza ejecutando varias veces el mismo algoritmo sobre la misma entrada
- el algoritmo de tirar la moneda no tiene esta propiedad, y por lo tanto su confianza no puede aumentarse ejecutándolo varias veces

- los algoritmos con esta característica (algunas respuestas siempre son correctas), se denominan sesgados, y permiten aumentar la confianza ejecutando varias veces el algoritmo sobre la misma instancia
- si el algoritmo es de decisión (respuesta **sí** o **no**), es sesgado, y es p-correcto, entonces con dos iteraciones se convierte en un algoritmo $1-(1-p)^2$ correcto

Algoritmo con mejora de la confianza

```
FUNCTION RepeatFreivalds (A,B,C,n,k)
 FOR i ::= 1 TO k
   IF no Freivalds(A,B,C,n)
     RETURN false
   ENDIF
 ENDFOR
 RETURN true
```

- la probabilidad de error de este nuevo algoritmo es $1-2^{-k}$, que coresponde a la probabilidad de que la fila de la diferencia no sea escogida k veces consecutivas

- cuando $k = 10$, la respuesta es 99,9% correcta
- se puede obtener así una certeza mayor que la de un algoritmo determinístico considerando probables errores de HW o SW
- si se quiere un error menor que ε, entonces el algoritmo lleva $\Theta(n^2 \log 1/\varepsilon)$

Verificación de Primalidad

- Problema: verificar si un número n es primo
- este problema tiene muchas aplicaciones, como por ejemplo en el sistema criptográfico RSA en la generación de claves
- el algoritmo determinístico directo es de $O(\sqrt{n})$ que lo hace inviable para números no muy grandes
- la siguiente propiedad podría generar un algoritmo probabilístico para este problema

Teorema 1 (Fermat)
Si $n \in \mathbf{N}$ es primo, entonces $a^{n-1} \,\mathbf{mod}\, n = 1$ para cualquier a tal que $1 \leq a \leq n-1$.

- la contrapositiva del teorema anterior sugiere que si encontramos un a que no cumple la propiedad, entonces el número no es primo
- se genera al azar un número $a, 1 < a \leq n-1$ y se calcular $a^{n-1} \,\mathbf{mod}\, n$
- el problema es que cuando el a buscado cumple la propiedad, no sabemos nada si el número es primo o no

Algoritmo

```
FUNCTION Fermat (n) % n>=4, impar
  a ::= uniforme(1..n-1)
  IF Expomod(a,n-1,n)=1
    RETURN true
  ELSE
    RETURN false
  ENDIF
```

- si el algoritmo dice que n es primo, entonces puede ser cierto o no
- en cambio, si el algoritmo dice que el n no es primo, entonces la respuesta siempre es correcta
- se trata de un algoritmo sesgado
- para poder aumentar la confianza en el algoritmo, es necesario ver que sea p-correcto, para algún p

- los testigos falsos de primalidad son los números a que hacen que el algoritmo devuelva **true** cuando en realidad n no es primo
- estos números son escasos: entre los primeros 1000 números enteros, sólo existen 4490 testigos falsos de primalidad sobre 172878 candidatos posibles
- esto resulta en una probabilidad de error menor al 3,3 %
- pero esto no significa que el algoritmo sea p-correcto para algún p (¿porqué?)

- no se puede afirmar que el algoritmo sea p-correcto para ningún p
- existen números como 651693055693681 tal que el 99,9965 % de los números menores son testigos falsos, a pesar de que es un número compuesto
- en este caso el algoritmo sólo da la respuesta correcta el 0,0035 % de las veces
- para cualquier p siempre existen contraejemplos que hacen que el algoritmo no se p-correcto

- afortunadamente, existe un teorema que, extendiendo el teorema de Fermat, proporciona una mejor manera de testear la primalidad.

Definición 2

Sea n un entero impar, $n \geq 4$. Existen $s > 0$, t impar tal que $n-1 = 2^s t$. Un número a, $2 \leq a \leq n-2$ se dice que pertenece a $B(n)$ si y solo si cumple una de las siguientes propiedades:

- $a^t \,\mathbf{mod}\, n = 1$, *o*
- *existe* i, $0 \leq i < s$ *tal que* $a^{2^i t} \,\mathbf{mod}\, n = n-1$.

Teorema 3

Sea $n > 4$, impar. Si n es primo entonces $B(n) = \{a \mid 2 \leq a \leq n-2\}$. Si n es compuesto, entonces $\mid B(n) \mid \leq (n-9)/4$.

- parece complicado ver si $a \in B(n)$ pero es fácil de implementar
- es cuestión de iterar sobre t desde 0 y aplicar exponenciación modular


```
FUNCTION Btest(n,a)
  s::=0; t::=n-1
  REPEAT
    s::=s+1; t::=t div 2
  UNTIL t mod 2=1
  x ::= expomod(a,t,n)
  IF  x=1 or x=n-1
    RETURN true
  ELSE
    FOR i::=1 TO s-1
      x::=x^2 mod n
      IF x=n-1 THEN RETURN true
    ENDFOR
  ENDIF
  RETURN false
```


- por ejemplo para ver si $158 \in B(289)$ primero se encuentran $t = 9$, $s = 5$ ya que $288 = 2^5 9$
- luego se encuentra $x = \texttt{expomod}(158, 9, 289) = 131$, y como no es 1 ni 288 se ejecuta el ciclo (a lo sumo 4 veces):

$$\begin{aligned}
a^{2t}\textbf{mod } n &= 131^2\textbf{mod } n = 110 \\
a^{2^2 t}\textbf{mod } n &= 110^2\textbf{mod } n = 251 \\
a^{2^3 t}\textbf{mod } n &= 251^2\textbf{mod } n = 288
\end{aligned}$$

- como el valor de la última iteración es $n-1$, entonces el ciclo para y el algoritmo devuelve **true**

- el teorema siguiente permite usar `Btest` para comprobar la primalidad
- y también da una cota a los posibles errores!

Teorema 4

Sea $n \geq 4$, impar. Entonces si n es primo, $B(n) = \{a | 2 \leq a \leq n-2\}$, y si n es compuesto, $|B(n)| \leq (n-9)/4$.

- el teorema anterior afirma que los testigos falsos fuertes de primalidad para todo número compuesto n son menos del 25% de los números entre 2 y $n-2$
- entonces, se podría pensar en un algoritmo probabilístico $0,75$-correcto, sesgado (la respuesta **false** es correcta), generando al azar números entre 2 y $n-2$, y controlando si pertenecen a $B(n)$

```
FUNCTION MillerRabin(n)    % n>=4 impar
 a ::= uniforme(2... n-2)
 RETURN Btest(n,a)
```


- entonces se puede aumentar la confianza iterando el algoritmo

```
FUNCTION RepetirMillerRabin(n,k)  % n>=4 impar
  FOR i::=1 TO k
    IF no MillerRabin(n)
      RETURN false
    ENDIF
  ENDFOR
  RETURN true
```


- resulta un algoritmo $(1-4^{-k})$-correcto, de tiempo $O(k\log^3 n)$
- con $k=10$ se tiene más certeza que la de un error de HW o SW.

Algoritmos Las Vegas

- los algoritmos Las Vegas se pueden clasificar a su vez en:
  - aquellos que siempre devuelven una respuesta correcta. La aleatoridad se usa para emparejar el costo de todas las instancias: se roba tiempo a las instancias eficientes para disminuir el tiempo de instancias ineficientes (efecto Robin Hood). Ejemplos: quicksort, selección, hashing.
  - aquellos que pueden reconocer que se han equivocado. El algoritmo toma decisiones al azar que eventualmente pueden ocasionar que no encuentra ninguna respuesta. En este caso se tiene la alternativa de volver a ejecutar el algoritmo. Ejemplo: ocho reinas, factorización de enteros

Factorización de enteros

- Problema: se $n > 1$ entero, se quiere encontrar la descomposición única de n como producto de números primos
- el problema de descomposición consiste en dado un número compuesto n, encontrar un divisor no trivial de n
- el problema de factorización se puede resolver mediante descomposición y testeo de primalidad: si n no es primo, encontrar un factor no trivial m, y recursivamente factorizar n/m y m

- el algoritmo trivial para descomposición es de $O(\sqrt(n))$

```
FUNCTION descomponer (n)
  FOR m ::= 2 TO piso(sqrt(n))
    IF n div m=0 THEN RETURN m
  ENDFOR
  RETURN n
```


- el tiempo de ejecución hace el algoritmo inviable aún para enteros medios: consideranto que cada iteración toma un nanosegundo, tomaría miles de años descomponer un entero difícil de 40 dígitos
- para que el entero sea difícil de descomponer, tiene que ser producto de dos primos de más o menos la misma longitud

Teorema 5

Sea n un número entero compuesto, a, b distintos enteros entre 1 y $n-1$ tal que $a+b \neq n$. Entonces si $a^2 \mathbf{mod} n = b^2 \mathbf{mod} n$, vale que $\gcd(a+b, n)$ es un divisor no trivial de n.

- el algoritmo consiste entonces en encontrar $1 < a, b < n-1$ tales que $a^2 \mathbf{mod} n = b^2 \mathbf{mod} n$ pero $a+b \neq n$, y usar el algoritmo de Euclides para encontrar $\gcd(a+b, n)$
- se puede ver que si n tiene al menos dos factores primos, entonces siempre existe al menos un par de números a, b con las propiedades mencionadas
- para encontrar los números a, b que se buscan, se generan x_i números aleatorios entre 1 y $n-1$ tales que su cuadrado módulo n tenga factores primos siempre menores o iguales que el k-ésimo primo
- se necesitan al menos $k+1$ de estos números
- se calculan sus cuadrados y se factorean, todo módulo n

- luego se buscan combinaciones de estos cuadrados de forma que todos los factores primos aparezcan con exponente par. Así es fácil calcular sus raíces: a se obtiene multiplicando todos los factores primos a la mitad del exponente; b se obtiene como el producto de los x_i generados que intervienen en la combinación
- se obtienen así a, b tales que $1 < a, b < n-1$ y $a^2 \mathbf{mod} n = b^2 \mathbf{mod} n$. Pero no se garantiza que $a \neq b$ o que $a + b \neq n$
- en estos casos, el algoritmo Las Vegas detecta el error
- pero en lugar de comenzar otra búsqueda de a, b desde el principio, simplemente se prueban otras combinaciones de sus cuadrados, o eventualmente se genera más x_i
- queda determinar el k para optimizar la performance, pero no es trivial

Ejemplo

- se quiere factorear $n = 2537$. Supongamos $k = 7$, o sea los factores primos de los números generados sólo pueden ser $2, 3, 5, 7, 11, 13, 17$
- se generan números x_i al azar, se obtiene la factorización de sus cuadrados módulo n, y se conservan sólo aquellos que tienen todos sus factores menores o iguales a 17
- observar que el proceso es razonable porque no es necesario encontrar todos los factores, tan pronto como se sabe que el número x_i tiene un factor mayor que 17, el número es descartado

- supongamos que los números encontrados son:

$$\begin{array}{rl}
x_1 = 2455 & x_1^2 \mathbf{mod} n = 1650 = 2 \times 3 \times 5^2 \times 11 \\
x_2 = 970 & x_2^2 \mathbf{mod} n = 2210 = 2 \times 5 \times 13 \times 17 \\
x_3 = 1105 & x_3^2 \mathbf{mod} n = 728 = 2^3 \times 7 \times 13 \\
x_4 = 1458 & x_4^2 \mathbf{mod} n = 2295 = 3^3 \times 5 \times 17 \\
x_5 = 216 & x_5^2 \mathbf{mod} n = 990 = 2 \times 3^2 \times 5 \times 11 \\
x_6 = 80 & x_6^2 \mathbf{mod} n = 1326 = 2 \times 3 \times 13 \times 17 \\
x_7 = 1844 & x_7^2 \mathbf{mod} n = 756 = 2^2 \times 3^3 \times 7 \\
x_8 = 433 & x_8^2 \mathbf{mod} n = 2288 = 2^4 \times 11 \times 13
\end{array}$$

- se encuentran combinaciones de los cuadrados de estos números de forma que todos los factores primos tengan exponente par
- una opción para esto es crear una matriz booleana de $(k+1) \times k$ y aplicar eliminación Gauss-Jordan en aritmética base 2.

- por ejemplo,

$$x_1^2 x_2^2 x_4^2 x_8^2 = 2^6 \times 3^4 \times 5^4 \times 11^2 \times 13^2 \times 17^2$$

- luego

$$\begin{array}{rcl}
a & = & (2^3 \times 3^2 \times 5^2 \times 11 \times 13 \times 17) \mathbf{mod} 2537 = \\
 & = & 2012 \\
b & = & (x_1 x_2 x_4 x_8) \mathbf{mod} 2537 = \\
 & = & (2455 \times 970 \times 1458 \times 433) \mathbf{mod} 2537 = 1127
\end{array}$$

- como $a \neq b$ y $a + b \neq 2537$ entonces para encontrar un factor de 2537 se obtiene $\gcd(a+b, n) = 43$ mediante el algoritmo de Euclides

- no todas las combinaciones generan números a, b válidos
- por ejemplo si $x_1^2 x_3^2 x_4^2 x_5^2 x_6^2 x_7^2 = 2^8 \times 3^{10} \times 5^4 \times 7^2 \times 11^2 \times 13^2 \times 17^2$
- entonces

$$\begin{aligned}
a &= (2^4 \times 3^5 \times 5^2 \times 7 \times 11 \times 13 \times 17)\mathbf{mod} 25370 \\
&= 1973 \\
b &= (x_1 x_3 x_4 x_5 x_6 x_7)\mathbf{mod} 2537 = \\
&= (2455 \times 1105 \times 1458 \times 216 \times 80 \times 1844)\mathbf{mod} 2537 = \\
&= 564
\end{aligned}$$

- como $a + b = 1973 + 564 = 2537$ el algoritmo Las Vegas debe retractarse y buscar otra combinación de los x_i, o generar más x_i y buscar nuevas combinaciones