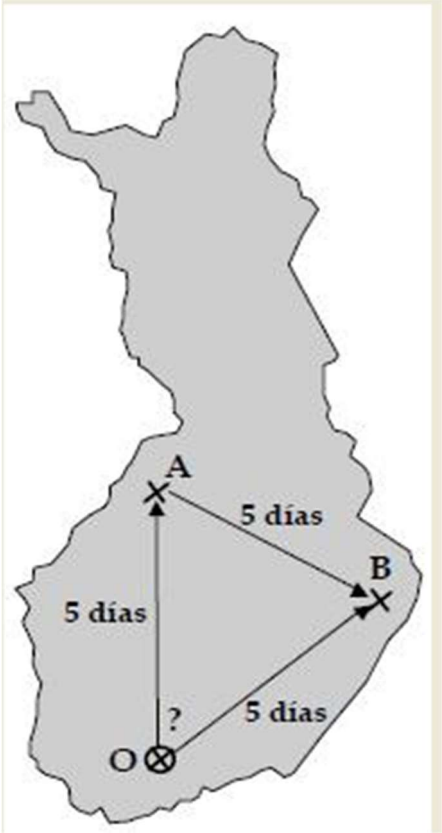
TEMA 6. ALGORITMOS NO DETERMINISTAS

1. Introducción.

Una historia sobre un tesoro, un dragón, un computador, un elfo y un doblón. En A o B hay un tesoro de x lingotes de oro pero no sé si está en A o B. Un dragón visita cada noche el tesoro llevándose y lingotes. Sé que si permanezco 4 días más en O con mi computador resolveré el misterio. Un elfo me ofrece un trato: Me da la solución ahora si le pago el equivalente a la cantidad que se llevaría el dragón en 3 noches.



¿Qué debo hacer? Si me quedo 4 días más en O hasta resolver el misterio, podré llegar al tesoro en 9 días, y obtener x-9y lingotes. Si acepto el trato con el elfo, llego al tesoro en 5 días, encuentro allí x-5y lingotes de los cuales debo pagar 3y al elfo, y obtengo x-8y lingotes. Es mejor aceptar el trato pero…¡hay una solución mejor! ¿Cuál? ¡Usar el doblón que me queda en el bolsillo! Lo lanzo al aire para decidir a qué lugar voy primero (A o B). Si acierto a ir en primer lugar al sitio adecuado, obtengo x-5y lingotes. Si no acierto, voy al otro sitio después y me conformo con x-10y lingotes. El beneficio esperado medio es x-7’5y.

Conclusión: en algunos algoritmos en los que aparece una decisión, es preferible a veces elegir aleatoriamente antes que perder tiempo calculando qué alternativa es la mejor. Esto ocurre si el tiempo requerido para determinar la elección óptima es demasiado frente al promedio obtenido tomando la decisión al azar. Esta es la idea básica de un algoritmo no determinista o probabilístico.

La característica fundamental de un algoritmo probabilísticos es que el mismo algoritmo puede comportarse de forma distinta aplicando los mismos datos. Las diferencias entre los algoritmos deterministas y probabilistas son las siguientes:

- A un algoritmo determinista nunca se le permite que no termine: hacer una división por 0, entrar en un bucle infinito, etc. A un algoritmo probabilista se le puede permitir siempre que eso ocurra con una probabilidad muy pequeña para datos cualesquiera. Si ocurre, se aborta el algoritmo y se repite su ejecución con los mismos datos.
- Si existe más de una solución para unos datos dados, un algoritmo determinista siempre encuentra la misma solución (a no ser que se programe para encontrar varias o todas). Un algoritmo probabilista puede encontrar soluciones diferentes ejecutándose varias veces con los mismos datos.
- A un algoritmo determinista no se le permite que calcule una solución incorrecta para ningún dato. Un algoritmo probabilista puede equivocarse siempre que esto ocurra con una probabilidad pequeña para cada dato de entrada. Repitiendo la ejecución un número suficiente de veces para el mismo dato, puede aumentarse tanto como se quiera el grado de confianza en obtener la solución correcta.
- El análisis de la eficiencia de un algoritmo determinista es, a veces, difícil. El análisis de los algoritmos probabilistas es, muy a menudo, muy difícil.

Un comentario sobre "el azar" y "la incertidumbre": a un algoritmo probabilista se le puede permitir calcular una solución equivocada, con una probabilidad pequeña. Un algoritmo determinista que tarde mucho tiempo en obtener la solución puede sufrir errores provocados por fallos del hardware y obtener una solución equivocada. Es decir, el algoritmo determinista tampoco garantiza siempre la certeza de la solución y además es más lento. Más aún, hay problemas para los que no se conoce ningún algoritmo (determinista ni probabilista) que dé la solución con certeza y en un tiempo razonable (por ejemplo, la duración de la vida del programador, o de la vida del universo…). Es mejor un algoritmo probabilista rápido que dé la solución correcta con una cierta probabilidad de error. Ejemplo: decidir si un nº de 1000 cifras es primo.

2. Algoritmos de Monte Carlo.

Sea p un número real tal que $0<p<1$.Un algoritmo de Monte Carlo es p–correcto si devuelve una solución correcta con probabilidad mayor o igual que p, cualesquiera que sean los datos de entrada. A veces, p dependerá del tamaño de la entrada, pero nunca de los datos de la entrada en sí.

Un problema muy relevante que se puede resolver con el método de Monte Carlo es el de determinar si un número es primo. Es el algoritmo de Monte Carlo más conocido. Este problema es especialmente útil para las técnicas criptográficas, que se basan en la factorización de números muy grandes en factores primos. No se conoce ningún algoritmo determinista que resuelva este problema en un tiempo razonable para números muy grandes (cientos de dígitos decimales). En los últimos años hemos asistido a una expansión sin precedentes en el uso de la criptografía de clave publica, incluso en el ámbito de las relaciones institucionales, como es el caso del DNI electrónico y del pasaporte electrónico. Estos documentos y otros muchos necesitan utilizar números primos (generalmente de gran tamaño) como elementos básicos para generar las claves o

para otros estadios del protocolo criptográfico. El algoritmo probabilístico más utilizado es el de Miller-Rabin, que se basa en el teorema menor de Fermat(1640):

Sea n un número primo. Entonces a^{n-1} mod n=1 para cualquier entero a tal que 1<=a<=n-1. Por ejemplo, sean n = 7 y a = 5. Entonces, $a^{n-1} = 5^6 = 15625 = 2232x\ 7 + 1$. Utilizaremos la versión contrapositiva de este teorema: Si n y a son enteros tales que 1<=a<=n y a^{n-1} mod n≠1, entonces n no es un número primo. Aunque no lo hemos visto, existe un algoritmo basado en la técnica Divide y Vencerás que permite realizar la exponenciación modular (resolver a^n mod z) que está en O(log n). Utilizando dos propiedades elementales de aritmética modular:

$$xy \bmod z = [(x \bmod z)\ (y \bmod z)] \bmod z$$
$$(x \bmod z)^y \bmod z = x^y \bmod z$$

tendríamos el siguiente pseudocódigo:

```
Function expomod(a, n, z) // calcula a^n mod z
        i=n
        r=1
        x=a mod z
        mientras i > 0 hacer
                si (i es impar) entonces
                        r = rx mod z
                        x =  x^2 mod z
                        i =ent(i /2)
                fif
        fmientras
        devolver r
ffun
```

El problema es que para verificar que un número n es primo, habrá que comprobar que todos los valores a entre 1 y n-1 cumplen que a^{n-1} mod n = 1. Sin embargo, podríamos probarlo con un solo número elegido al azar entre 1 y n-1. Por ejemplo, una primera versión del algoritmo probabilista es:

```
Function Fermat(n)
        a = uniforme(1..n-1)
        si expomod(a,n-1,n)=1 entonces devolver True
        sino devolver False
fin fun
```

Cuando Fermat(n) devuelva falso estaremos seguros de que n no es primo. Sin embargo, si Fermat(n) devuelve cierto no podemos concluir nada. Este ejemplo demuestra la forma de trabajar de los algoritmos tipo Monte Carlo.

Otro ejemplo de aplicación de los algoritmos de Montecarlo es determinar si un vector tiene un elemento mayoritario. Para ello, se comprueba al azar si alguno de esos elementos aparece un número mayor de N/2 veces. El algoritmo sería así:

```
Función MayoritarioMonteCarlo(v, N)
        x=v[aleatorio(1…N)]
        total=0
        desde i=1 hasta n hacer
                si (v(i)=x) entonces total=total+1
        fdesde
        devolver (total>N/2)
ffun
```

Para los vectores no mayoritarios este método tiene probabilidad 1 de acertar (ya que ningún elemento se repite más de N/2 veces). Para los vectores que sí tienen un elemento mayoritario, el método acierta con probabilidad mayor que 1/2, exactamente con probabilidad p=(#apariciones del elemento mayoritario)/N. Luego falla con Pf<1/2. No está mal. Se podría mejorar haciendo:

```
Función MayoritarioMonteCarlo2(v, N)
        Si MayoritarioMonteCarlo(v, N) entonces
                Devolver Verdad
        si no
                Devolver MayoritarioMonteCarlo(v, N)
```

Ahora suponiendo que el vector sea mayoritario solo fallará si MayoritarioMonteCarlo falla dos veces. Por tanto, MayoritarioMonteCarlo2 falla con probabilidad $p<(1/2)\cdot(1/2)=1/4$. Para generalizarlo con una cota de error $\varepsilon>0$:

```
Función MayoritarioMonteCarlo_ε(v, N, e)
        p=1
        m=False
        repetir
                p=p/2
                m=MayoritarioMonteCarlo(v,N)
        hasta m o (p<ε/2)
        devolver m
```

3. Algoritmos de Las Vegas.

Un algoritmo de Las Vegas, se diferencia de uno de Montecarlo, en que nunca da una solución falsa, sino que toma decisiones al azar para encontrar una solución antes que un algoritmo determinista y si no encuentra solución lo admite. Hay dos tipos de algoritmos de Las Vegas, atendiendo a la posibilidad de no encontrar una solución:

a) Los que siempre encuentran una solución correcta, aunque las decisiones al azar no sean afortunadas y la eficiencia disminuya.
b) Los que a veces, debido a decisiones desafortunadas, no encuentran una solución.

El primer tipo se aplica a problemas en los que la versión determinista es mucho más rápida en el caso promedio que en el caso peor. Por ejemplo, en Quicksort, donde el coste peor es n^2 y el y coste promedio es nlogn. Los algoritmos de las Vegas pueden reducir o eliminar las diferencias de eficiencia para distintos datos de entrada. Con mucha

probabilidad, los casos que requieran mucho tiempo con el método determinista se resuelven ahora mucho más deprisa. En los casos en los que el algoritmo determinista sea muy eficiente, se resuelven ahora con más coste. En el caso promedio, no se mejora el coste, por lo que se realiza una uniformización del tiempo de ejecución para todas las entradas de igual tamaño.

El segundo tipo es aceptable si fallan con probabilidad baja. En ese caso se vuelven a ejecutar con los mismos datos de entrada. Se utilizan para resolver problemas para los que no se conocen algoritmos deterministas eficientes que los resuelvan. El tiempo de ejecución no está acotado, pero es razonable con una elevada probabilidad. Se presentan en forma de procedimiento con una variable éxito que toma valor cierto si se obtiene solución y falso en otro caso. Sería LV(x, solución, éxito) donde x representa los datos de entrada.

Sea p(x) la probabilidad de éxito si la entrada es x. Los algoritmos de Las Vegas exigen que p(x)>0 para todo x. Sea la siguiente función:

```
Function repetirLV(x)
        repetir
                LV(x, solución, éxito)
        hasta éxito
        devolver solución
ffun
```

El número de pasadas del bucle es 1/p(x). Sea t(x) el tiempo esperado de repetirLV(x). La primera llamada a LV tiene éxito al cabo de un tiempo s(x) con probabilidad p(x). La primera llamada a LV fracasa al cabo de un tiempo f (x) con probabilidad 1- p(x). El tiempo esperado total en este caso es f (x) + t(x), porque después de que la llamada a LV fracase volvemos a estar en el punto de partida (y por tanto volvemos a necesitar un tiempo t(x)). Así, t(x) = p(x)s(x) + (1-p(x))(f (x) + t(x)). Resolviendo la ecuación anterior se obtiene:

$$t(x) = s(x) + \frac{1-p(x)}{p(x)} f(x)$$

Esta ecuación es la clave para optimizar el rendimiento de este tipo de algoritmos.