Chapter 9

Algoritmos probabilistas.

9.1 Introducción.

A veces puede ocurrir que cuando un algoritmo afronta una decisión posible entre varias posibilidades, es preferible seleccionar aleatoriamente alguna de esas posibilidades en lugar de perder tiempo en estudiar cual de esas alternativas es la mejor. Esta situación se presenta cuando el tiempo que se necesitaría para determinar cual de las alternativas es la mejor, es prohibitivo en comparación con el tiempo de análisis que se ahorra al seleccionar una alternativa al azar.

Debido a esta característica, un mismo algoritmo se puede comportar de forma distinta cuando se aplica dos veces a un mismo problema. Su tiempo de ejecución e incluso los resultados obtenidos pueden variar ostensiblemente. Esta posibilidad da pie a que puedan tener cierta ventaja frente a algoritmos deterministas. Por ejemplo, en un algoritmo determinista no se puede permitir un bucle infinito o una división por cero, ya que ese algoritmo nunca podría resolver el caso en que eso se produjese. Sin embargo esto si es admisible en un algoritmo probabilista siempre y cuando esos casos no permitidos se produzcan con una probabilidad muy baja. Si esa situación anómala se produjese en un algoritmo probabilista bastaría con interrumpirlo y ejecutarlo de nuevo, cosa que no daría resultado en un algoritmo determinista. Otra ventaja consiste en que si hay más de una solución correcta, ejecutando el algoritmo varias veces podríamos obtener esas soluciones. Esto también se podría hacer con un algoritmo determinista, pero habría que implementarlo para obtener todas las soluciones.

Otra consecuencia que se deriva del comportamiento aleatorio es que a veces se permitirá que produzcan resultados erróneos, siempre y cuando esto suceda con una probabilidad muy baja y que se pueda invocar al algoritmo varias veces para que se pueda tener la certeza de que se ha obtenido una respuesta correcta. Un algoritmo determinista que produjese resultados erróneos para ciertos datos de entrada es inadmisible, ya que siempre fallará para esas entradas.

Muchos algoritmos probabilistas dan respuestas probabilistas, es decir, que no son necesariamente exactas. En determinados tipos de aplicaciones críticas no se pueden permitir respuestas inciertas, sin embargo es posible que la probabilidad de error se pueda reducir por debajo del error que se cometería por el equipo físico durante el mayor tiempo que necesita para obtener una solución determinista. Ello implica que puede ocurrir que un algoritmo probabilista, de una respuesta incierta que no sólo se obtenga más rápido, sino que además es más fiable que la que se obtienen de un algoritmo determinista. Por otra parte, hay que destacar que existen problemas en los que no se conoce ningún algoritmo, ya sea determinista o probabilista, que pueda dar una respuesta fiable en una cantidad razonable de tiempo y sin embargo existen algoritmos probabilistas que pueden resolver rápidamente el problema si se admite una pequeña probabilidad de error.

Hay dos clases de algoritmos probabilistas que no garantizan la corrección del resultado.

- Los *algoritmos numéricos* producen un intervalo de confianza de la forma *con una probabilidad del 95% la solución es 55 más o menos 2*. Cuanto más tiempo empleen estos algoritmos, más preciso es el intervalo. Estos algoritmos pueden ser más eficientes, aunque menos precisos que los deterministas y son útiles si nos basta con una aproximación de la respuesta correcta.

- Los *algoritmos de Monte Carlo* dan la respuesta exacta con una alta probabilidad, aunque pueden proporcionar una respuesta incorrecta. Normalmente no se puede saber si un algoritmo de este tipo da una respuesta correcta, pero se puede reducir la probabilidad de error dando al algoritmo más tiempo.

No todos los algoritmos probabilistas dan respuestas probabilistas. En muchas ocasiones las opciones probabilistas se usan como un medio para acelerar la búsqueda de la solución deseada, en cuyo caso la respuesta obtenida siempre es correcta. Además tambien podría ocurrir que la corrección de toda posible solución se pueda verificar eficientemente. Los algoritmos probabilistas que nunca dan una respuesta incorrecta se denominan de *Las Vegas*. Si cabe la posibilidad de que el algoritmo falle, no ocurre nada, basta con seguir aplicando el algoritmo a la entrada usada hasta que tenga éxito.

Un ejemplo para aclarar el funcionamiento de estos tipos de algoritmos probabilistas podría ser un algoritmo para obtener el año en el que los musulmanes entraron en España. La versión numérica si fuese invocada 5 veces obtendría por ejemplo los siguientes resultados:

- Entre 708 y 717.

- Entre 710 y 719.

- Entre 705 y 714.

- Entre 700 y 709.
- Entre 702 y 711.

Dándole más tiempo al algoritmo, se podría reducir la probabilidad de obtener un intervalo falso (en este caso el 20%) o de reducir la amplitud de los intervalos (en este caso 10), o ambas cosas a la vez. La versión de Monte Carlo si se invocase 8 veces podría responder: $711, 710, 711, 711, 711, 212, 711, 711$, que produce una tasa de error del 25%. Esta tasa se podría reducir dando más tiempo al algoritmo. En este caso se puede ver que a veces las respuestas erróneas son muy cercanas a la correcta y en otras son muy dispares. Por último, la versión de Las Vegas produciría una salida tal como ésta: $711, 711, error, 711, 711, error, 711, 711$. El algoritmo no da nunca una respuesta incorrecta, pero a veces no responde.

Otro aspecto importante a tener en cuenta en los algoritmos probabilistas es el estudio de los tiempos de ejecución. En los algoritmos deterministas se definía el *tiempo promedio* de un algoritmo como el tiempo promedio de que requiere un algoritmo para todos los casos de un tamaño dado, considerando que todos los casos son equiprobables. Este tiempo promedio puede inducir a error si todos los casos no son equiprobables. El *tiempo esperado* de un algoritmo probabilista se define para cada caso individual y sería el tiempo promedio para ese caso si se resolviese varias veces. También se puede hablar de *tiempo esperado promedio* y *tiempo esperado en el caso peor* de un algoritmo probabilista. Este último se refiere al tiempo que requiere el peor caso posible de un tamaño dado y no al tiempo requerido si se tomasen las peores opciones probabilistas.

9.2 Algoritmos probabilistas numéricos.

La aleatoriedad se empleó por primera vez en algorítmica para resolver problemas numéricos. Una de las aplicaciones más extendidas es la simulación. Mediante la simulación se pueden obtener soluciones aproximadas a problemas en los que es complicado obtener soluciones por métodos deterministas. Para ciertos problemas en la vida real el cálculo de una solución exacta no es posible. Ello puede ser debido a incertidumbres en los datos experimentales que se usen, porque el ordenador maneje solo datos enteros o racionales, mientras que la solución es irracional o porque se tarde demasiado tiempo en dar una respuesta precisa.

La respuesta que ofrece un algoritmo de este tipo es siempre aproximada, pero esta precisión puede aumentarse a medida que aumenta el tiempo de ejecución del algoritmo. El error suele ser inversamente proporcional a la raíz cuadrada de la cantidad de tiempo que se haya empleado, así que se necesita aumentar cien veces el tiempo empleado para aumentar la precisión un dígito.

9.2.1 La aguja de Buffon.

En el siglo XVIII, Georges Louis Leclerc, conde de Buffon, demostró que si se arroja aleatoriamente una aguja de longitud l, de forma tal que caiga aleatoriamente sobre un punto cualquiera y su orientación también sea aleatoria, sobre un suelo que tenga planchas rectangulares unidas de ancho $2l$, la probabilidad de que la aguja caiga sobre dos planchas es de $1/\pi$. La figura 9.1 ilustra un ejemplo usando 12 agujas. De esta forma, se puede predecir que si lanzamos n agujas, aproximadamente $\frac{n}{\pi}$ ocuparán dos planchas.

Otra aplicación posible del experimento podría ser la estimación del valor de π. Si lanzamos N agujas y suponiendo que n han caído sobre dos planchas, aplicando el teorema de Buffon, tendremos que $n = \frac{N}{\pi}$ de donde se decuce que $\pi = \frac{N}{n}$. Si queremos obtener estimaciones precisas de π habrá que dejar caer muchas más agujas.

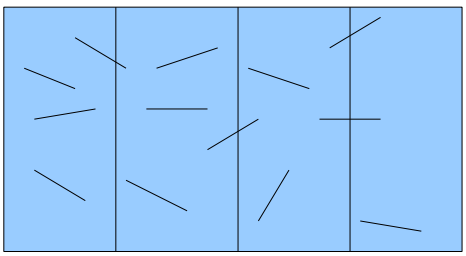

Figure 9.1: Ejemplo de experimento de la Aguja de Buffon

Aunque existen algoritmos deterministas mucho más precisos y eficientes para determinar experimentalmente el valor de π, este fue muy usado en el siglo XIX y fue uno de los primeros algoritmos probabilistas usados.

El algoritmo podría plantearse como sigue:

Algoritmo $Buffon(N;;\pi)$
  inicio
    $n \leftarrow 0$
    para i **de** 1 **a** N **hacer**
      $aguja = generarAguja()$
      si $ocupaDosPlanchas(aguja) = cierto$ **entonces**
        $n \leftarrow n + 1$
      finsi

finpara
$\pi \leftarrow \frac{N}{n}$
fin

Buffon también demostró que si L es la anchura de las planchas y l la longitud de las agujas, la probabilidad de que una aguja caiga entre dos planchas es $\frac{2l}{L\pi}$. De esta forma, también sería posible estimar experimentalmente el ancho de las planchas.

Estos algoritmos convergen a la respuesta correcta a medida que el número de agujas usadas tiende a ∞, sin embargo lo hacen muy lentamente, y se necesita dejar caer la aguja 100 veces más que las realizadas hasta un momento dado, para poder aumentar un dígito la precisión.

9.2.2 Integración numérica.

Este es el más conocido de los algoritmos probabilistas numéricos, también denominado integración de Monte Carlo, aunque no es un ejemplo de *Algoritmo de Montecarlo* según la clasificación que hemos visto. Sea $y = f(x)$ una función contínua en el intervalo $[a, b]$ y de valor positivo en todo el intervalo, entonces la superficie delimitada por la función, las rectas verticales $x = a$, $x = b$ y el eje X es:

$$I = \int_a^b f(x)dx$$

Si se considera un rectángulo cuya base la determinen los puntos $(a, 0)$ y $(b, 0)$ y cuya altura sea $\frac{I}{b-a}$, la superficie de dicho rectángulo será la misma que la citada anteriormente. Dado que ambos tienen la misma superficie, ambos han de tener la misma altura media. Por lo tanto la altura media de la curva $y = f(x)$ en el intervalo $[a, b]$ es $\frac{I}{b-a}$. La figura 9.2 ilustra este método.

A partir de aquí es relativamente sencillo implementar un algoritmo probabilista para estimar dicha superficie. Lo único que habría que hacer es estimar la altura media de la función $f(x)$ por muestreo aleatorio y multiplicar dicha altura media por $b - a$. El algoritmo quedaría como sigue:

Algoritmo $IntegracionNumerica(N, f, a, b; ; superficie)$
inicio
$suma \leftarrow 0$
para i **de** 1 **a** N **hacer**
$x \leftarrow uniforme(a, b)$ `Devuelve un número aleatorio entre a y b`
$suma \leftarrow suma + f(x)$
finpara
$superficie \leftarrow (b - a)\frac{suma}{N}$
fin

De manera similar al algoritmo de la aguja de Buffon, se necesita 100 veces más simulaciones para aumentar la precisión un dígito. Es por ello que cualquier

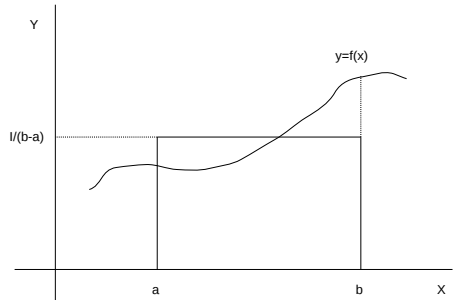

Figure 9.2: Integración numérica

algoritmo determinista, como pueda ser el de los trapecios es mucho más eficiente y preciso.

Un caso en el que el algoritmo probabilista pueda ser más eficiente, es el que se presenta a la hora de obtener una integral múltiple. En este caso, cualquier algoritmo determinista, tendría que hacer un muestreo de las variables independientes correspondientes a cada una de las dimensiones, con lo cual el número de puntos muestreados crece exponencialmente. Si tenemos cuatro dimensiones, y tres variables independientes, y tomamos 100 puntos por variable, tendríamos una muestra de 1000000 de puntos. Si se integra usando el algoritmo citado la dimensión de la integral suele tener poco efecto en la precisión obtenida. En general esta integración de Monte Carlo se emplea para evaluar integrales de 4 o más dimensiones.

9.3 Algoritmos de Monte Carlo.

Existen problemas para los cuales no se conoce un algoritmo eficiente (probabilista o determinista) que sea capaz de ofrecer una solución correcta en todas las ocasiones. Los algoritmos de Monte Carlo cometen ocasionalmente un error, pero encuentran la solución correcta con una probabilidad alta, independientemente del caso estudiado. Esta afirmación es más contundente que decir que el algoritmo funciona correctamente en la mayoría de los casos, fallando tan solo de vez en cuando en algunos casos especiales, ya que aquí se afirma que no hay ningún caso en el cual la probabilidad de error es elevada. El problema es que no queda constancia cuando el algoritmo comete un error, es decir, no avisa.

Se dice que un algoritmo de Monte Carlo es $p - correcto$ si devuelve una respuesta correcta con una probabilidad no menor que p, sea cual sea el caso consider-

ado. En algunos casos, p puede depender del tamaño del caso, pero nunca del caso en sí. Una característica de estos algoritmos es que se puede reducir arbitrariamente la probabilidad de error a costa de un ligero aumento del tiempo de cálculo. Esta característica se denomina *amplificación de la ventaja estocástica*.

A continuación se detallan algunos ejemplos prácticos de este tipo de algoritmos.

9.3.1 Verificación de la multiplicación de matrices.

Se tienen tres matrices A, B y C cuadradas de orden n y se sospecha que $C = AB$. Se puede comprobar si es cierto o no, realizando el producto de ambas y comparando el resultado con C. El producto normal consume un tiempo $O(n^3)$ y los algoritmos más sofisticados están el el orden $O(n^{2.37})$. La pregunta está en si se podría mejorar dicho tiempo admitiendo una pequeña probabilidad de error. Se puede demostrar que para cualquier $\epsilon > 0$ prefijado, se necesita un tiempo de $O(n^2)$ para resolver el problema con una probabilidad de error menor que ϵ.

Suponiendo que $C \neq AB$, sea $D = AB - C$. De dicha forma D no es nula. Sea i un entero que representa una fila de D que contiene al menos un elemento no nulo. Consideremos un subconjunto S del conjunto $\{1, 2, 3, \ldots, n\}$. Supongamos que $\sum_S D$ representa al vector que se obtiene sumando componente a componente las filas de D, cuyos índices coinciden con los valores almacenados en S. Supongamos que para generar el subconjunto S recorremos todas las filas $1, 2, 3, \ldots, n$ y lanzamos una moneda para cada fila, para ver cual de ellas se introduce en S. dado que más o menos existe la misma probabilidad de que la fila i pertenezca a S como que no pertenezca, la probabilidad de que $\sum_S D \neq 0$ es como mínimo de $\frac{1}{2}$.

Por otra parte, $\sum_S D$ siempre es 0 si $C = AB$. De aquí se deduce comprobar si $C = AB$ calculando $\sum_S D$ para un subconjunto S seleccionado aleatoriamente, y comparando con 0 el resultado. El problema estriba en calcular esto de forma eficiente sin calcular primero D y por tanto AB. La solución consiste en considerar $\sum_S D$ como una multiplicación de matrices independiente. Sea X un vector binario de n elementos tal que $X(j) = 1$ si $j \in S$ y $X(j) = 0$ si $i \notin S$. Se puede verificar facilmente que $\sum_S D = XD$ considerando a X una matriz de 1 fila y n columnas. De esta forma la comprobación se restringe a verificar si $X(AB - C)$ es o no cero, lo que equivale a que $XAB = XC$ para algún vector binario seleccionado arbitrariamente. En principio parece que calcular XAB puede requerir más tiempo que calcular AB, pero esto no es así si primero se multiplica XA obteniendo una nueva matriz de una fila y n columnas, con lo cual el producto es mucho más rápido. De esta forma este producto se reduce a un tiempo $O(n^2)$.

El algoritmo quedaría como sigue:

Algoritmo $VerificacionMultiplicacionMatrices(A, B, C, n; ;)$
inicio

```
    para i de 1 a n hacer
      x(i) ← uniforme(0,1) Devuelve un número aleatorio que vale
0 o 1
    finpara
    si (XA)B = XC entonces
      devolver cierto
    sino
      devolver falso
    finsi
  fin
```

A la vista de lo anterior, se puede considerar que el algoritmo devuelve una respuesta correcta con una probabilidad del 50% en todos los casos, por lo tanto es $\frac{1}{2}$-correcto por definición. Sin embargo no es $p-$correcto para todo p menor que $\frac{1}{2}$, ya que la probabilidad de error es de $\frac{1}{2}$ cuando C difiere de AB precisamente en una fila, ya que se devuelve una respuesta incorrecta si y solo si $X(i) = 0$, donde la fila i supone la diferencia entre C y AB. Esto no supone ninguna ventaja, ya que lanzando una moneda, viendo su resultado podríamos decidir si es falso o no sin necesidad de analizar las tres matrices en cuestión. La única ventaja es que cuando el algoritmo devuelve falso, tenemos plena certeza de que la respuesta es correcta, sin embargo si devuelve verdadero puede ser que la respuesta no sea correcta.

Una mejora de la fiabilidad consistiría en aplicar el algoritmo varias veces sobre el mismo caso, si al menos en una ocasión se obtiene el resultado falso, se puede concluir que $AB \neq C$ independientemente de las veces que obtengamos verdadero. Este algoritmo quedaría como sigue:

```
Algoritmo RepetirVerificacionMultiplicacionMatrices(A, B, C, n, k; ; )
  inicio
    para i de 1 a k hacer
      si VerificacionMultiplicacionMatrices(A, B, C, n) = falso entonces
        devolver falso
      finsi
    finpara
    devolver cierto
  fin
```

Para ver la fiabilidad de este algoritmo se pueden analizar dos casos. Si realmente $AB = C$ siempre devolverá *cierto*. En este caso la probabilidad de error será 0. Por otra parte si $C \neq AB$ entonces la probabilidad de que una llamada al algoritmo base devuelva *cierto* es como máximo $\frac{1}{2}$. Si realizamos varias llamadas estas probabilidades se van multiplicando, ya que las llamadas serían independientes unas de otras. Por tanto, la probabilidad de que devuelva un valor erróneo después de k llamadas sería de $\frac{1}{2^k}$ como máximo. en este caso el último algoritmo sería $(1 - \frac{1}{2^k})-$correcto. Cuando $k = 10$ la probabilidad de acertar es mayor de 0.999. Este decrecimiento en el error es típico en los algoritmos de Monte Carlo

que resuelven problemas de decisión, la respuesta ha de ser cierto o falso y ha de estar garantizada su certeza.

Este algoritmo se podría implementar dando una cota superior a la probabilidad de error admisible:

Algoritmo $CotaErrorVerificacionMultiplicacionMatrices(A, B, C, n, \epsilon; ;)$
  inicio
    $k \leftarrow log\frac{1}{\epsilon}$
    devolver $RepetirVerificacionMultiplicacionMatrices(A, B, C, n, k)$
  fin

En esta versión se puede analizar su tiempo de ejecución como función simultánea del tamaño del caso y de la probabilidad de error.

9.4 Algoritmos de Las Vegas.

En este caso se toman decisiones probabilistas para acelerar la búsqueda de una solución correcta. A diferencia de los de Monte Carlo, nunca proporcionan una respuesta incorrecta. Existen dos tipos:

- Los que usan la aleatoriedad para guiar su búsqueda de forma que se garantice una solución correcta, aunque se tomen decisiones poco eficientes, ya que si esto ocurre el algoritmo solo necesitaría emplear más tiempo.

- Los que pueden tomar caminos equivocados, que conducen a caminos sin salida, haciendo imposible la obtención de una solución en esta ejecución del algoritmo.

Los del primer tipo suelen usarse cuando un algoritmo determinista conocido para resolver el problema en cuestión es mucho más rápido en el caso promedio que en el caso peor. Esto ocurre en el quicksort. Su tiempo promedio es $O(nlogn)$ y su tiempo peor es $O(n^2)$. Incorporando un elemento aleatorio se permite reducir y a veces eliminar esta diferencia entre casos buenos y malos.

Puede ocurrir que el análisis de la eficiencia del caso promedio induzca a error, ya que depende de la distribución de probabilidad de los casos a manejar. En el quicksort si los datos se distribuyen aleatoriamente el tiempo de ejecución sería $O(nlogn)$ y en caso de que estén casi ordenados el tiempo sería de orden $O(n^2)$, con lo cual el algoritmo es vulnerable a una distribución de probabilidad inesperada que puede ocurrir en un caso particular. Los algoritmos de Las Vegas subsanan este inconveniente, e igualan el tiempo empleado para diferentes casos de un tamaño dado.

Los algoritmos de Las Vegas no son mejores en tiempo promedio que los algoritmos deterministas. Los casos que requieren mucho tiempo con los deterministas

si son acelerados usando algoritmos de Las Vegas pero aquellos aquellos casos que requieren de un mínimo tiempo con el algoritmo determinista se enlentecen hasta un valor medio cuando se usa un algoritmo de Las Vegas.

En el otro tipo de vez en cuando se toman decisiones que llevan a un callejón sin salida, en cuyo caso el algoritmo reconoce su error. Esta posibilidad de fallo es inadmisible en un algoritmo determinista, sin embargo es admisible en un algoritmo de Las Vegas siempre y cuando la probabilidad de que esto ocurra no sea muy alta. Cuando se produzca un fracaso se aplica nuevamente el algoritmo para ese caso y así tener la posibilidad de éxito.

Un algoritmo de las vegas tendrá el siguiente prototipo $LasVegas(x;;y,exito)$ donde x es el caso a resolver, $exito$ indica si el algoritmo ha tenido éxito o no y y es la solución siempre y cuando $exito = cierto$. Para que un algoritmo sea de Las Vegas se exige que la probabilidad de éxito para un caso x cualquiera sea mayor de 0. Esto asegura que encontrará una solución siempre si se sigue repitiendo el algoritmo.

El algoritmo de repetición hasta obtener una solución correcta sería el que sigue:

Algoritmo $RepetirLasVegas(x)$
  inicio
    repetir
      $LasVegas(x;;y,exito)$
    hasta que $exito = cierto$
    devolver y
  fin

9.4.1 El problema de las 8 reinas (versión 2).

La solución a este problema se puede obtener usando backtracking, como se vió en el tema correspondiente. Sin embargo, como se pudo apreciar en el algoritmo planteado, el número de nodos que se van generando en el árbol es muy grande y además las posiciones que ocupan las reinas en las soluciones no siguen ningún patrón y parecen estar dispuestas aleatoriamente. Esta característica sugiere usar un algoritmo voraz de Las Vegas que vaya poniendo aleatoriamente reinas en filas sucesivas, teniendo en cuenta que no se puede poner una que sea amenazada por una reina colocada previamente. Al ser voraz, no se intentan reubicar las reinas anteriores cuando se llega a un callejón sin salida. En este caso el algoritmo tendrá éxito cuando ubica las ocho reinas o fracasará cuando llega a un callejón sin salida. La aproximación voraz facilita la implementación del algoritmo, aunque sea más largo de implementar, pero también implica una probabilidad de fracaso.

A continuación se detalla el algoritmo $n - reinasLasVegas$ que también hará uso del algoritmo $Lugar()$ que se detalló en el tema del Backtracking.

El algoritmo en pseudocódigo sería el siguiente:

Algoritmo $n-reinasLasVegas(n;;X,exito)$
inicio
Inicializamos la solución a 0
para i **de** 1 **a** n **hacer**
$x(i) \leftarrow 0$
finpara
//Se han colocado k-1 reinas y se buscan todas las posiciones para la k-ésima
para k **de** 1 **a** n **hacer**
$contador \leftarrow 0$
Almacena todas las posiciones posibles la reina k en el vector ok
para j **de** 1 **a** n **hacer**
$X(k) \leftarrow j$ *probamos la reina k en la columna j*
si $Lugar(k,x) = cierto$ **entonces** *Se puede colocar en la columna j*
$contador \leftarrow contador + 1$
$ok(contador) \leftarrow j$ *guardamos la posicion disponible*
finsi
finpara
salir si $contador = 0$ *no se ha encontrado posicion para la reina k*
Se puede colocar la reina k y se selecciona una posición aleatoria
$columna \leftarrow ok(uniforme(1, contador))$
$X(k) \leftarrow columna$
finpara
si $contador = 0$ **entonces**
$exito \leftarrow falso$ *No hay solución*
sino
$exito \leftarrow cierto$ *Hay solución*
finsi
fin

Algoritmo $Lugar(k,x;;)$
inicio
para i **de** 1 **a** n **hacer**
si $(x(i) = x(k)$ **o** $|x(i) - x(k)| = |i-k|$ **entonces**
devolver $falso$
finsi
finpara
devolver $cierto$
fin

Se puede demostrar que por término medio se necesitan 8 pruebas para obtener

una solución correcta.