Análisis de Algoritmos y Estructuras de Datos

Práctica 1: Generación de ejemplares de prueba

Versión 1.2

1. Introducción

Tanto durante la comprobación de la corrección de un programa, como durante el análisis de su eficiencia, es importante poder generar ejemplares de prueba aleatorios. La base fundamental de esta técnica reside en nuestra capacidad de generar números pseudoaleatorios por medio de una computadora. Veamos por qué.

Para intentar detectar posibles errores en un programa, puede construirse una batería de pruebas «de caja negra». Idealmente, esta batería está formada por una gran cantidad de parejas (e, s) que constan de un ejemplar o dato de entrada y de su correspondiente resultado o salida. Se intenta cubrir el mayor número posible de casos: normalmente se incluyen casos especiales, que son en los que más fácil resulta errar al programar, y pruebas exhaustivas para algunos casos generales que sigan un determinado patrón.

Por ejemplo, la siguiente función de ordenación[1] contiene un oscuro error de programación: compila perfectamente, siempre termina su ejecución y nunca provoca errores durante la misma. Pero, a pesar de todo, contiene un error, ya que hemos constatado que no siempre devuelve un vector ordenado.

```
void ordena(int* p, int* f)
{
  if (f - p > 1 ) {
    int* k = p + (f - p) / 2;
    ordena(p, k - 1);
    ordena(k, f);
    inplace_merge(p, k, f);
  }
}
```

Es fácil probar casos aislados: hemos creado la siguiente función que comprueba si al ordenar un vector, e, con nuestra función se obtiene otro vector dado, s. Esta función, también muestra el ejemplar en cuestión en la salida estándar.

[1] No se preocupe si en estos momentos no entiende su definición: no es de lo que se trata aquí.

```
bool prueba_ordena(int e[], int s[], unsigned n)
{
  int v[n];

  for (unsigned i = 0; i < n; ++i) {
    v[i] = e[i];              // Copia la entrada en un vector auxiliar
    cout << v[i] << " ";      // y lo muestra en cout.
  }
  cout << endl;
  ordena(v, v + n);           // «Ordena» el vector.
  return (memcmp(v, s, n) == 0); // Devuelve la comprobación del resultado.
}
```

Sólo es cuestión de suministrar a esta función los ejemplares de entrada y los resultados esperados hasta encontrar un error. Si lo encontramos. Nunca se insiste lo suficiente en que la principal limitación de este enfoque es su incapacidad para demostrar que un programa es correcto, salvo que el número de posibles entradas sea finito y además sea factible comprobar todas ellas. Lo más que podremos demostrar es que contiene errores, si damos con alguno.

Nos gustaría encontrar un ejemplar que reprodujera este error y que fuera lo suficientemente pequeño como para poder aislar el problema y corregirlo. Podemos automatizar el proceso creando un programa que genere exhaustivamente todas las permutaciones hasta una cierta longitud, vaya probando la función y nos informe de si encuentra un fallo.

Para ello, podemos usar un algoritmo genérico de la biblioteca de C++, *next_permutation*() (declarado en <*algorithm*>), que genera cíclicamente la siguiente permutación de una secuencia en orden lexicográfico. Este algoritmo devuelve falso si ha producido la permutación ordenada y verdadero en caso contrario. Por lo tanto, si se emplea reiteradamente comenzando por una permutación ordenada, generará todas las posibles hasta llegar otra vez a la inicial.

Teniendo en cuenta las consideraciones anteriores, el programa principal queda como sigue:

```
int main()
{
  const unsigned N = 8;
  int v[N],       // Permutación probada
      w[N],       // Resultado esperado
      i = 0;      // Contador de pruebas

  for (int n = 1; n <= N; ++n) {
    v[n - 1] = n - 1;         // Permutación actual ordenada
    memcpy(w, v, sizeof v); // Vector ordenado
    do {
      cout << setw(5) << ++i << ": ";
      if (!prueba_ordena(v, w, n)) {     // ¿Hay un error?
        cout << "ERROR." << endl;
        return 1;
      }
    } while (next_permutation(v, v + n));    // Siguiente permutación
  }
```

```
  }
  return 0;
}
```

Hemos tenido suerte y, en este caso, se encuentra rápidamente que la secuencia 1, 0, 2, 3, produce un error. Pero no siempre tendremos tanta suerte y hay que tener en cuenta, si deseamos hacer una prueba exhaustiva, que el número de permutaciones crece muy rápidamente con la longitud de la secuencia. Los ejemplares de prueba aleatorios pueden ser útiles para completar una batería de pruebas de caja negra como la anterior y cubrir casos de mayor tamaño.

También es muy común el empleo de ejemplares aleatorios durante el análisis empírico de un algoritmo. Por ejemplo, en el caso de algoritmos de ordenación por comparación, es habitual realizar el análisis en el promedio bajo la hipótesis de equiprobabilidad de todas las permutaciones de una secuencia de entrada determinada. Un análisis experimental riguroso basado en dicha hipótesis requiere ser capaz de generar permutaciones aleatorias uniformemente distribuidas para su utilización como datos de entrada.

2. Números pseudoaleatorios

La función *rand*(), declarada en <*cstdlib*>, no recibe parámetros y devuelve un entero pseudoaleatorio entre 0 y *RAND_MAX*. Si se emplea *rand*() reiteradamente, se obtiene una secuencia de números pseudoaleatorios que, en realidad, está determinada por un valor inicial denominado *semilla*. Es por ello que los números obtenidos se denominan «pseudoaleatorios», en lugar de «aleatorios».

La función *srand*() recibe un único parámetro (un entero sin signo) y se encarga de establecer dicho valor como semilla para *rand*(). Inicialmente, la semilla ya tiene un valor adecuado, 1, por lo que normalmente no hay que preocuparse de emplear esta función.

De la explicación anterior se deduce que las secuencias producidas por *rand*() serán idénticas si se generan a partir de una misma semilla, lo que puede ser muy útil cuando se quiera asegurar la repetibilidad de un experimento aleatorio.

Si por alguna buena razón no se desea este comportamiento, puede hacerse depender la semilla de alguna cantidad variable. Esto se logra, normalmente, utilizando como semilla algo externo al programa y que varíe de una ejecución a otra.

Por ejemplo, en el siguiente programa, se emplea como semilla el valor devuelto por *time*(), que varía a cada segundo. Por lo tanto, es harto improbable que el valor de la semilla sea el mismo, siempre que entre dos ejecuciones del programa medie al menos un segundo.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;
```

```
int main()
{
  srand(time(nullptr));
  for (int i = 0; i < 10; ++i)
    cout << rand() << endl;
}
```

3. Permutaciones pseudoaleatorias

La función *random_shuffle*(), declarada en <*algorithm*>, forma parte de la biblioteca estándar de C++ e implementa un algoritmo genérico que podemos utilizar con un vector, por ejemplo, para obtener una permutación aleatoria de los elementos almacenados en todo o parte del vector.

Por ejemplo, tras ejecutar el siguiente fragmento, quedará en el vector v una de las $8! = 40\,319$ permutaciones (sin repetición) que se pueden formar con los 8 primeros números naturales:

```
int v[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
const size_t n = sizeof v / sizeof *v;
random_shuffle(v, v + n);
```

Además, la función *random_shuffle*() tiene la propiedad de que cualquiera de las posibles permutaciones tiene la misma probabilidad de aparecer, es decir, las permutaciones que se obtendrán siguen una distribución uniforme.[2]

No obstante, tenga siempre presente que esto también se basa en la generación de números pseudoaleatorios: a igual semilla, igual secuencia de resultados.

Ejercicios

1. Utilizando *rand*(), escriba una función que genere un número pseudoaleatorio entero en el intervalo discreto $[a, b]$.

2. Escriba un programa que simule 10 000 000 de tiradas de un dado. Imprima las frecuencias relativas que se obtienen para cada una de las caras. Emplee la función del ejercicio anterior.

3. Utilizando *rand*(), escriba una función que genere un número pseudoaleatorio de coma flotante y precisión doble en el intervalo continuo $[a, b]$.

4. Escriba un programa que genere 10 000 000 de números pseudoaleatorios de precisión doble en el intervalo $[0, 1]$ e imprima la media de los valores $y = 4 \cdot \sqrt{1 - x^2}$ correspondientes a cada número x. Emplee la función del ejercicio anterior.

5. Escriba un programa que genere 10 000 000 de permutaciones pseudoaleatorias de los 6 primeros números naturales e imprima cuántas de ellas están ordenadas.

[2] En realidad, para que esto sea así, el generador de números pseudoaleatorios subyacente debe generar una distribución uniforme y poseer un periodo lo suficientemente grande.