Laboratorio 3 de Programación 3

Leandro Rabindranath León

Este laboratorio es evaluado y la calificación que obtengas cuenta para tu calificación definitiva.

Fecha de entrega: martes 5 de mayo de 2015 hasta la las 6pm

1. Introducción

El fin de este laboratorio es desarrollar destrezas algorítmicas con el uso de listas. Para cumplir este propósito en esta laboratorio instrumentaremos un programa que resuelva un rompecabezas llamado sudoku.

Para la realización de este laboratorio requieres:

1. La suite clang
2. La biblioteca $\aleph_\omega$ (Aleph-ω); especialmente debes estar familiarizado con el tipo `DynList` y sus distintos métodos.

1.1. El juego de sudoku

El sudoku es un rompecabezas en el cual se plantea una matriz 9×9 para colocar números entre 1 y 9. Inicialmente la matriz contiene algunos números iniciales de base. La idea es completar las celdas vacías de modo que para cada celda i, j se satisfagan las siguientes condiciones:

1. La fila i no puede contener números repetidos.
2. La columna j no puede contener números repetidos
3. La submatriz 3×3 donde se encuentra i, j no puede contener números repetidos.

   La matriz 9×9 del sudoku se divide en 9 submatrices cuadradas de 3×3. Para que el sudoku esté válido, ninguna de sus submatrices puede contener números repetidos.

1.2. Representación del sudoku

En este laboratorio representaremos un sudoku mediante una lista de listas de enteros del siguiente modo:

```
typedef DynList<DynList<int>> Sudoku;
```

Los valores de la matriz están comprendidos entre 0 y 9. Un valor distinto es un error.

El valor 0 representa que la celda no tiene un número asignado. Es decir, que falta por asignarle un número entre 0 y 9.

Las celdas del sudoku se referencian desde la esquina superior izquierda. Por ejemplo, para el sudoku (no resuelto)

```
-------------------------
| 4 0 0 | 1 0 3 | 0 0 0 |
| 0 0 8 | 0 7 0 | 0 0 0 |
| 0 0 9 | 0 0 0 | 6 7 0 |
-------------------------
| 0 9 0 | 0 1 0 | 0 0 7 |
| 7 0 0 | 2 0 8 | 0 0 5 |
| 6 0 0 | 0 4 0 | 0 1 0 |
-------------------------
| 0 3 2 | 0 0 0 | 4 0 0 |
| 0 0 0 | 0 3 0 | 1 0 0 |
| 0 0 0 | 8 0 4 | 0 0 9 |
-------------------------
```

El valor $0, 0$ es 4 y el $8, 8$ es 9. No olvides que los índices comienzan en cero.

El cuadrado

```
---------
| 0 9 0 |
| 7 0 0 |
| 6 0 0 |
---------
```

Está situado en la posición $3, 0$

El método `nth(i)` de `DynList` recibe una posición de acceso `i` y retorna el elemento situado en esa posición. Así, si tenemos un `sudoku` y queremos examinar el valor de la celda i, j basta con ejecutar:

```
sudoku.nth(i),nth(j)
```

Para conocer -y eventualmente modificar- su valor.

1.3. Representación en archivo texto

Los sudokus pueden entrarse al solucionador mediante un archivo texto. Cada línea representa una fila de la matriz. Por ejemplo:

```
1 8 0 0 0 0 0 2 9
6 0 2 5 0 0 0 3 0
5 0 0 8 0 2 6 0 0
3 0 0 0 0 5 2 0 6
0 0 0 2 0 7 0 0 0
```

```
4 0 5 1 0 0 0 0 7
0 0 9 4 0 1 0 0 8
0 5 0 0 0 9 1 0 2
8 4 0 0 0 0 0 5 3
```

1.4. Los archivos fuente

1.4.1. `sudoku.H`

Contiene definiciones del sudoku y las funciones utilitarias:

1. `void print_sudoko(const Sudoku & sudoku)`

   Imprime por consola el sudoku.

2. `string sudoku_array(const Sudoku & sudoku)`

   Imprime por consola un sudoku representado como arreglos.

3. `void print(const DynList<int> & l)`

   Imprime por consola el contenido de la lista `l`.

¡No modifiques este archivo!

1.4.2. `Sudoku.C`

Contiene las rutinas que instrumentarás en este laboratorio.
Tu entrega consistirá en este archivo.

1.4.3. `sudoku-solve.C`

Resuelve un sudoku entrado por archivo. Por ejemplo:

```
./sudoku-solve s2.txt
```

Produce como salida:

```
-------------------------
| 1 8 0 | 0 0 0 | 0 2 9 |
| 6 0 2 | 5 0 0 | 0 3 0 |
| 5 0 0 | 8 0 2 | 6 0 0 |
-------------------------
| 3 0 0 | 0 0 5 | 2 0 6 |
| 0 0 0 | 2 0 7 | 0 0 0 |
| 4 0 5 | 1 0 0 | 0 0 7 |
-------------------------
| 0 0 9 | 4 0 1 | 0 0 8 |
| 0 5 0 | 0 0 9 | 1 0 2 |
```

```
| 8 4 0 | 0 0 0 | 0 5 3 |
-------------------------
-------------------------
| 1 8 4 | 6 7 3 | 5 2 9 |
| 6 7 2 | 5 9 4 | 8 3 1 |
| 5 9 3 | 8 1 2 | 6 7 4 |
-------------------------
| 3 1 7 | 9 4 5 | 2 8 6 |
| 9 6 8 | 2 3 7 | 4 1 5 |
| 4 2 5 | 1 6 8 | 3 9 7 |
-------------------------
| 2 3 9 | 4 5 1 | 7 6 8 |
| 7 5 6 | 3 8 9 | 1 4 2 |
| 8 4 1 | 7 2 6 | 9 5 3 |
-------------------------
```

O sea, el sudoku entrado y su solución

No es necesario modificar este archivo. Se te provee para que pruebes tu solución definitiva con sudokus reales.

1.4.4. `sudoku-tests`

Contiene una batería de pruebas de todos las rutinas que deberás implementar.

Tu evaluación será realizada según una variante de este archivo.

Revísalo y estúdialo. Aparte de que comprenderás cómo serás evaluado, el archivo contiene técnicas que te pueden ser útiles.

Todos los tests de este programa están comentados a efector de no generar errores de ejecución. Descomentalos en la medida en que vayas progresando y probando.

2. Laboratorio

El laboratorio está dividido en dos partes: validación de sudoku y resolución.

2.1. Validación

Un aspecto sumamente importante de la algorítmica es operar sobre entradas ideales. Un algoritmo se diseña para resolver un problema bajo condiciones de entrada válidas. Es un error de diseño incorporar al algoritmo la validación de la entrada.

Así las cosas, la primera parte de este laboratorio está consagrada a escribir rutinas que nos permitan asegurar que el sudoku sea válido.

2.1.1. Repitencia de elementos

Programa

```
bool check(const DynList<int> & l)
```

Debe retornar `true` si la lista `l` contiene exactamente 9 elementos no repetidos entre 1 y 9. De lo contrario se retorna `false`.

2.1.2. Igualdad de listas

Programa

```
bool lists_equal (const DynList<int> & l1, const DynList<int> & l2)
```

La cual retorna `true` si la lista `l1` es idéntica a la lista `l2`.

2.1.3. Obtención de fila

Escribe:

```
DynList<int> sudoku_row(const Sudoku & sudoku, int row)
```

La cual retorna una lista con el contenido de la fila `row` del `sudoku`.

2.1.4. Obtención de columna

Escribe:

```
DynList<int> sudoku_col(const Sudoku & sudoku, int col)
```

La cual retorna una lista con el contenido de la columna `col` del `sudoku`.

2.1.5. Verificación de fila

Escribe

```
bool check_row(const Sudoku & sudoku, int row)
```

La cual retorna `true` si la fila `row` de `sudoku` es válida. Es decir, contiene 9 valores, no repite los números y éstos están dentro de $[1, 9]$.

2.1.6. Verificación de filas

Escribe

```
bool check_rows(const Sudoku & sudoku)
```

La cual retorna `true` si todas las filas de `sudoku` son válidas.

2.1.7. Verificación de columna

Escribe

```
bool check_col(const Sudoku & sudoku, int col)
```

La cual retorna `true` si la columna `col` de `sudoku` es válida. Es decir, contiene 9 valores, no repite los números y éstos están dentro de $[1, 9]$.

2.1.8. Verificación de columnas

Escribe

```
bool check_cols(const Sudoku & sudoku)
```

La cual retorna `true` si todas las columnas de `sudoku` son válidas.

2.1.9. Obtención de cuadrado

Escribe

```
DynList<int> sudoku_square(const Sudoku & sudoku, int row, int col)
```

La cual retorna una lista con los valores de la submatriz 3×3 situada en `i,j`. Revisa los tests de `sudoku-solve` si tienes dificultades para comprender.

2.1.10. Verificación de cuadrado

Escribe

```
bool check_square(const Sudoku & sudoku, int row, int col)
```

La cual retorna `true` si el cuadrado situado en `row,col` es válido. Es decir, contiene 9 valores, no repite los números y éstos están dentro de $[1, 9]$.

2.1.11. Verificación de cuadrados

Escribe

```
bool check_squares(const Sudoku & sudoku)
```

La cual retorna `true` si todos los cuadrados de `sudoku` son válidos.

2.1.12. Verificación de un sudoku no resuelto

Programa

```
bool check_well_formed(const Sudoku & sudoku)
```

La cual retorna `true` si `sudoku` está "bien formado"; es decir, contiene 9 filas, cada fila contiene 9 elementos y todos los elementos están dentro de $[0, 9]$. **Nota que puesto que el sudoku pudiera no estar resuelto el cero** 0 **es permitido como valor dentro de una celda**.

2.1.13. Verificación de sudoku

Programa

```
bool check_sudoku(const Sudoku & sudoku)
```

El cual retorna `true` si `sudoku` es válido; es decir, si todas sus filas, columnas y cuadrados son válidos.

Cuando un sudoku esté resuelto, entonces esta rutina debe retornar `true`.

2.1.14. Búsqueda de elemento en lista

Programa

```
bool list_has(const DynList<int> & l, int val)
```

La cual retorna `true` si `val` está dentro de la lista `l`.

2.1.15. Compáración de sudokus

Escribe

```
bool are_equals(const Sudoku & s1, const Sudoku & s2)
```

La cual retorna `true` si los sudokus `s1` y `s2` son idénticos.

2.2. Resolución

Las rutinas anteriores están orientadas a validar sudokus y a detectar cuando estén resueltos.

Ahora en esta parte programarás rutinas que pueden ayudarte a resolver un sudoku.

2.2.1. Prueba de lista

Escribe

```
bool ok_for_list(const DynList<int> & l, int val)
```

La cual retorna `true` si es posible meter en la lista `l`, que contiene valores cero, el valor `val` (entre 1 y 9) sin que éste se repita.

2.2.2. Prueba de fila

Escribe

```
bool ok_for_row(const Sudoku & sudoku, int row, int val)
```

La cual retorna `true` si es posible meter `val` el la fila `row` de `sudoku`.

2.2.3. Prueba de columna

Escribe

```
bool ok_for_col(const Sudoku & sudoku, int col, int val)
```

Que retorna `true` si es posible meter `val` en la columna `col` de `sudoku`.

2.2.4. Prueba de cuadrado

Escribe

```
bool ok_for_square(const Sudoku & sudoku, int row, int col, int val)
```

La cual retorna `true` si es posible meter `val` en el cuadrado situado en `row,col`.

2.2.5. Obtención de cuadrado según una coordenada

Programa

```
tuple<int, int> cell_to_square(int row, int col)
```

La cual retorna una tupla contentiva de la coordenada en fila y columna del cuadrado donde se sitúa `row,col`.

Ejemplos:

```
cell_to_square(5, 4) ----> (3,3)
cell_to_square(8, 8) ----> (6,6)
cell_to_square(3, 8) ----> (6,6)
```

Para crear una tupla de dos enteros `i,j` haz

```
tuple<int,int> tupla = make_tuple(i, j);
```

Dada una `tupla` sus elementos se acceden mediante:

```
int i = get<0>(tupla;
int j = get<1>(tupla);
```

2.2.6. Prueba de celda

Programa

```
bool ok_for_cell(const Sudoku & sudoku, int row, int col, int val)
```

La cual retorna `true` si es posible meter `val` en la celda del `sudoku` `row,col`.

2.2.7. Lista de candidatos según celda

Programa

```
DynList<int> avail_numbers(const Sudoku & sudoku, int row, int col)
```

La cual retorna una lista de todos los números que es posible meter en la celda del `sudoku` `row,col`.

2.2.8. Listas de celdas restantes por resolver

Ahora considera los tipos:

```
typedef tuple<int, int> Cell;

typedef DynList<Cell> Cells;
```

El primer tipo es una "celda" del sudoku consistente de una tupla cuyo primer elemento es una fila y segundo elemento una columna. El segundo tipo es una lista de celdas.

Programa

```
Cells avail_cells(const Sudoku & sudoku)
```

La cual retorna la lista de celdas disponibles para colocar números en el `sudoku`. Dicho de otro modo, retorna la lista de celdas con valor cero (0).

2.2.9. Lista de celdas a lista de enteros

A efectos de reusar código, especialmente la comparación entre listas, puede ser conveniente representas una lista de celdas como una lista plana. En ese sentido, programa la rutina:

```
DynList<int> cells_to_list(const Cells & cells)
```

La cual recibe la lista de celdas `cells` y retorna una lista plana de enteros correspondientes a las celdas. Los elementos en las posiciones pares contienen filas y en la impares columnas. El orden de la lista resultado debe ser el mismo que el de la lista `cells`

2.2.10. Solución del sudoku

Con todo lo que has programado hasta el presente, tienes instrumentada toda la maquinaria para resolver un sudoku.

Aquí debes programar

```
bool solve(Sudoku & sudoku, const DynList<Cell> & cells)
```

Antes que nada nota que esta es la única función de este laboratorio donde un `sudoku` es mutable (en todas las rutinas anteriores era inmutable).

La rutina recibe un `sudoku` y una lista de celdas disponibles para jugar. Retorna `false` si el `sudoku` no tiene solución. De lo contrario, la rutina resuelve el `sudoku`, asigna a las celdas disponibles su solución y retorna `true`.

Hay varias maneras de atacar el problema. Probablemente la más simple sea probar a fuerza bruta todas las combinaciones posibles.

Dado un `sudoku`, la rutina `aval_cells()` te retorna las celdas a las cuales aún no se le asignado un número. Así que el primer paso es seleccionar una de estas celdas. Luego, para esta celda, obtienes las posibles jugadas mediante `avail_number()`. Con esta información, juegas, es decir, asignas a la celda en cuestión uno de los números disponibles y luego invocas recursivamente a `solve()` pero con una celda de menos.

Asegúrate de restaurar la celda si `solve()` retorna `false`.

2.2.11. Solución final

Programa

```
Sudoku solve(const Sudoku & sudoku)
```

La cual retorna la solución del `sudoku`.

Si el `sudoku` no está bien formado, entonces se debe disparar la excepción `domain_error("Sudoku is not well formed")`. Si el `sudoku` no tiene solución entonces se debe disparar la excepción `domain_error("sudoku es unsolvable")`.

Esta rutina también sirve de base para la llamada inicial a `solve()` recursivo. Nota que en esta rutina el `sudoku` es inmutable, razón por la cual debes efectuar una copia del `sudoku` y pasarle esa copia al `solve()` recursivo.

3. Recomendaciones

1. Este laboratorio es un poco largo por la cantidad de rutinas que debes programar, pero no es difícil. La mayorías de las rutinas se pueden programar en una sola instrucción funcional. La rutina más larga y compleja de nuestra solución de referencia es `solve()` recursivo, el cual consume siete (7) líneas de código.

   La gran ganancia es que sabrás de listas. Pero quizá la pérdida es que no te provoque más resolver sudokus (ya tienes un programa que lo hace por ti).

2. Aunque las rutinas pueden escribirse muy cortas, no te preocupes si una o más te salen más largas. ¡Estás aprendiendo!

3. Puesto que el laboratorio es largo, empieza cuanto antes. No esperes hasta último momento.

4. Lo más aconsejable es resolver los ejercicios en el mismo orden sugerido.

5. ¡Reusa código! Muchos ejercicios se resuelven usando rutinas que ya programaste.

6. ¡Usa las pruebas! Guíate por `sudoku-tests`. Descomenta los tests en el orden en que vayas programando.

   Pero **no asumas que los tests garantizan que tu solución esté correcta.**. Añade tests si lo consideras conveniente.

7. ¡Usa el foro! Plantea preguntas y dudas. Está permitido discutir y explicar. **Lo que no está permitido es compartir código.**

   Especialmente, pregunta y discute por las rutinas funcionales de $\aleph_\omega$ (Aleph-ω), las cuáles no están completamente documentadas.

4. Evaluación

La evaluación será efectuada automáticamente por un programa que otorgará un puntuación y reportará los tests que no hayan sido satisfechos. La salida del programa será publicada.

Fecha tope de entrega: martes 5 de mayo hasta a las 6 pm. (sin prórroga)

Enviar el archivo `sudoku.C` a la dirección

`leandro.r.leon@gmail.com`

Con el subject, **imperativo y exacto, LAB-3-PR3**. **Debes poner en el cuerpo del email los nombres, apellidos y número de cédula de la pareja que somete el laboratorio.**

5. Trabajo ad hoc

El sudoku tiene otras variantes que podrías explorar en caso de que desees profundizar su comprensión y mejorar tus conocimientos.

5.1. Distintas soluciones

Un suduku puede tener más de una solución. En este sentido, puedes comenzar por explorar un algoritmo simple que te responda si el sudoku dado tiene o no más de una solución.

Luego puede explorar encontrar todas las soluciones posibles, lo cual con el algoritmo a fuerza bruta es simple: después de que encuentres la primera solución, continua la recursión con el resto de los números disponibles.

5.2. Otros tipos de sudoku

Una forma general de sudoku es con n números posibles, con matrices $n \times n$ y cuadrados $\sqrt{n} \times \sqrt{n}$. Por ejemplo, podrías manejar 16 números en matrices 16×16 y cuadrados 4×4. En este caso, aunque el esquema de solución es el mismo, puede ser de interés pensar en estructuras de datos más rápidas y esquemas de solución más eficientes que el de fuerza bruta.