

Trabajo Práctico N° 2

Diseño, Implementación y Análisis C++

75.04 Algoritmos y Programación II
Año 2018 - 1^{er} cuatrimestre



Alumno	Padrón	Email
Douce Suarez, Cristian G.	89.902	cristiandouce@yahoo.com.ar
Dreiling, Augusto	86.909	augusto.dreiling@gmail.com

Entrega: -

Calificación/Correcciones:

Índice

1. Enunciado	4
1.1. Objetivos	4
1.2. Alcance	4
1.3. Requisitos	4
1.4. Descripción	4
1.4.1. Diseño OO	4
1.4.2. Análisis de complejidad de los algoritmos	4
1.4.3. Análisis de complejidad de programa completo	4
1.4.4. Pruebas de performance	4
1.4.5. Testing	5
1.4.6. Troubleshooting	5
1.4.7. Portabilidad	5
1.4.7.1. Compiladores	5
1.4.7.2. Sistemas operativos	5
1.4.8. Interfaz	5
1.4.8.1. Formato de los archivos de entrada y salida	5
1.4.8.2. Línea de comando	6
1.4.8.3. Regresiones	6
1.4.9. Ejemplos	6
1.4.10. Informe	7
1.4.11. Fechas	7
2. Informe	8
2.1. Diseño e implementación	8
2.1.1. Ubicación de código fuente	8
2.1.2. Estructura del código fuente	8
2.1.3. Vendors / Librerías de la cátedra	9
2.1.4. Fuentes de los autores	9
2.2. Compilación del programa	9
2.3. Análisis de complejidad de los algoritmos	10
2.3.1. DFT	10
2.3.2. FFT	11
2.4. Análisis de complejidad del programa completo	12
2.4.1. Procesamiento de lectura y almacenamiento	12
2.4.2. Procesamiento de escritura	13
2.4.3. Procesamiento conjunto del programa	13
2.5. Pruebas de performance	14
2.6. Troubleshooting	17
2.6.1. Memory Leak Detection	17
2.6.2. Static Analyzer	18
2.7. Análisis y conclusiones de resultados	18
2.8. Código fuente	19
2.8.1. main.cpp	19
2.8.2. src/fourier.h	23
2.8.3. src/fourier.cpp	24
2.8.4. src/dft.h	27
2.8.5. src/dft.cpp	28
2.8.6. src/idft.h	29
2.8.7. src/idft.cpp	29
2.8.8. src/fft.h	30
2.8.9. src/fft.cpp	30
2.8.10. src/iff.h	32
2.8.11. src/iff.cpp	33
2.8.12. src/utlis.h	33

2.8.13. src/utils.cpp	34
2.8.14. src/program.h	36
2.8.15. src/program.cpp	36
2.8.16. vendor/cmdline.h	36
2.8.17. vendor/cmdline.cpp	37
2.8.18. vendor/complejo.h	41
2.8.19. vendor/complejo.cpp	41
2.8.20. vendor/lista.h	44
2.8.21. test/tests.sh	51
2.8.22. test/files/test _{1s} signals _{10000p} oints.txt	53
2.8.23. test/files/test _{10s} signals _{1000p} oints.txt	53
2.8.24. test/files/test _{100s} signals _{100p} oints.txt	56
2.8.25. test/results/perf-total.txt	60
2.8.26. test/results/perf-lectura.txt	60
2.8.27. test/results/perf-lectura-y-alg.txt	61
2.8.28. Makefile	62
2.8.29. Makefile.WIN32	62

1. Enunciado

1.1. Objetivos

Diseñar, implementar, y poner a prueba un programa en C++ que permita calcular la transformada rápida de Fourier (*FFT*, por sus siglas en inglés). Ejercitar conceptos de patrones, programación orientada a objetos, análisis de algoritmos, performance, testing, troubleshooting y portabilidad de programas.

1.2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

1.3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la sección 1.4.10, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

1.4. Descripción

En este trabajo continuamos nuestras implementaciones previas de la transformada de Fourier (*TP0*, *TP1*), procurando generar un programa con las siguientes características adicionales:

1.4.1. Diseño OO

El programa deberá adoptar un diseño orientado a objetos utilizando el patrón *strategy* (como hicimos en el *TP1*). Adicionalmente, la implementación de objetos deberá estar alineado con las pautas de diseño explicadas en clase, y con el feedback provisto en los trabajos anteriores.

1.4.2. Análisis de complejidad de los algoritmos

El informe deberá incluir un análisis de complejidad temporal y espacial de los algoritmos implementados (*DFT*, *FFT*).

1.4.3. Análisis de complejidad de programa completo

De forma similar, cada grupo deberá realizar un análisis de complejidad temporal y espacial del programa completo: es decir, incluyendo la complejidad vinculada a los procesos de entrada y salida de información.

1.4.4. Pruebas de performance

Reporte de las pruebas de performance realizadas, prestando especial atención a las predicciones de complejidad detalladas en los puntos anteriores.

Mediante estas pruebas, buscamos comprender cómo escalan los tiempos de ejecución a medida que se incrementa la longitud de la secuencia de entrada.

El informe deberá incluir además un análisis de la composición (apertura) de los tiempos de ejecución del programa, que permita responder las siguientes preguntas:

- ¿Qué fracción del tiempo se invierte en leer la entrada del programa?
- ¿Qué fracción del tiempo es usada para transformar o antitransformar esa información?
- ¿Qué fracción del tiempo se lleva la generación de la salida por pantalla y archivos?

1.4.5. Testing

El programa deberá permitir ejecutar casos de regresión automatizados como se explica a continuación.

Usando las opciones `-r` (archivo de regresiones) y `-e` (umbral de error, sección 1.4.8.2), el programa deberá comparar cada una de las secuencias de salida generadas con las líneas respectivas del archivo de regresión, calculando el módulo del vector de error relativo de la siguiente manera:

$$e_X = \sqrt{\frac{\sum |X[k] - R[k]|^2}{\sum |R[k]|^2}} \quad (1)$$

En donde X representa al vector de salida (calculado por el programa), mientras que R es el vector de regresión (leído del archivo pasado con `-r`), y e_X es el valor del error calculado para esa salida.

Para cada una de las regresiones ejecutadas, el programa deberá imprimir una línea por `std :: cout` indicando el resultado de la regresión, comparando cada valor e_X con el umbral de error (opción `-e`). Por ejemplo:

```
$ cat input.txt
1 0 0 0
1 -1 1 -1
$ cat regressions.txt
(4, 0) (0, 0) (0, 0) (0, 0)
(0, 0) (0, 0) (4, 0) (0, 0)
$ tp2 -i input.txt -r regressions.txt
test 1: ok 4 1.3e-12
test 2: ok 4 1.2e-12
```

Por último, en este caso, el programa deberá finalizar con código 0 sólo cuando todos los casos del archivo de regresión han generado un resultado satisfactorio (dentro del umbral de error relativo).

El resto de los detalles de la interfaz de línea de comando y entrada/salida, están detallados en la sección 1.4.8.

```
$ tp2 -i input.txt -o /dev/null -r regressions.txt
$ echo $?
0
```

1.4.6. Troubleshooting

Deberá usarse el módulo *memcheck* de **Valgrind** para analizar posibles leaks de memoria.

Deberá utilizarse alguna herramienta de análisis estático como *cppcheck* o *CoverityC++* para detectar errores en el código fuente del trabajo práctico.

1.4.7. Portabilidad

El trabajo deberá poder correr en múltiples plataformas, compiladores y sistemas operativos.

1.4.7.1. Compiladores

Al menos 3 de los siguientes: *llvm*, *gcc*, *VisualStudio*, *IntelC++*, *DigitalMarsC++*.

1.4.7.2. Sistemas operativos

El programa deberá soportar **Linux** y, adicionalmente, al menos 1 de los siguientes: **Windows**, **FreeBSD**.

1.4.8. Interfaz

1.4.8.1. Formato de los archivos de entrada y salida

En este trabajo adoptaremos el formato de los archivos que usamos durante el *TP1*.

1.4.8.2. Línea de comando

A la interfaz de línea de comando del TP anterior, le agregaremos 2 opciones necesarias para poder ejecutar las regresiones:

- $-r$, o $--regression$, ubicación del archivo con el contenido de las regresiones. Este archivo contiene, en cada línea, la salida precalculada correspondiente con la entrada pasada en $-i$. Tiene el mismo formato que los usados en las opciones $-i$ y $-o$. Cuando esta opción está presente, el programa deberá ejecutar las regresiones y generar una salida de acuerdo a lo explicado en la sección 1.4.5. En caso contrario, cuando la opción no está explícitamente en la línea de comando, el programa deberá comportarse en forma normal, de acuerdo a lo explicado en el TP1.
- $-e$, o $--error$, para indicar el umbral del error relativo máximo a tolerar durante la ejecución de las regresiones. El valor por defecto de este parámetro es $1e-3$.

1.4.8.3. Regresiones

Como vimos antes, la opción $-r$ activa la ejecución de las regresiones. En este caso, la salida del programa deberá consistir, exclusivamente, de una línea de texto por cada secuencia de datos procesada (línea del archivo de entrada).

Para cada línea, el programa deberá imprimir

1. La palabra test seguida del número de secuencia de la prueba (comenzando con 1 para la primera regresión, luego 2, y así sucesivamente)
seguido de:
2. el resultado de ejecución de la prueba (**ok** o **error**)
3. la longitud del vector transformado (potencia entera de 2)
4. la magnitud del error relativo de acuerdo a la ecuación en la sección 1.4.5, expresada en notación científica.
5. *newline* (**n**)

Durante la ejecución de las regresiones, el código de salida del programa deberá reflejar el resultado global de la ejecución del conjunto de regresiones: deberá ser 0 cuando todas las regresiones poseen un error menor al valor del umbral, y 1 cuando al menos una de las regresiones tenga un valor de error relativo igual o mayor al umbral controlado por la opción $--error$.

A continuación veremos algunos ejemplos de ejecución de casos, y del formato de salida a utilizar durante la ejecución de las regresiones.

1.4.9. Ejemplos

Se incluye para el trabajo práctico un conjunto de archivos de texto con señales y sus transformadas en el formato requerido para éste.

El ejemplo más simple consiste en una entrada vacía. Observar que la salida es también vacía:

```
$ touch entrada1.txt
$ ./tp1 -i entrada1.txt -o salida1.txt
$ cat salida1.txt
$
```

Los siguientes ejemplos son de transformadas simples. Observar que el primer uso del programa no especifica qué algoritmo usar, de manera que se tomará por defecto la *FFT*:

```
$ cat entrada2.txt
1 1 1 1
$ ./tp1 < entrada2.txt
(4, 0) (0, 0) (0, 0) (0, 0)
```

A continuación, vamos a generar el archivo de regresiones usando la implementación de la *DFT*, y luego ejecutar las regresiones sobre el algoritmo *FFT*:

```
$ ./tp1 -m dft -o regressions2.txt < entrada2.txt
$ ./tp1 -m fft -r regressions2.txt < entrada2.txt
test 1: 4 0
$ echo $?
0
```

Similarmente, para anti-transformar:

```
$ cat entrada3.txt
(0, 0) (0, 0) (4, 0) (0, 0)
$ ./tp1 -m ifft < entrada3.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
$ ./tp1 -m idft -o salida3.txt < entrada3.txt
$ cat salida3.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
```

El siguiente ejemplo es análogo al anterior, aunque su entrada no tiene una cantidad de puntos potencia de 2. Notar que el programa implícitamente extiende la entrada con ceros hasta alcanzar la potencia de 2 más cercana (que en este caso es 4):

```
$ cat entrada4.txt
(0, 0) (0, 0) (4, 0)
$ cat regressions4.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
$ ./tp1 -m idft -r regressions4.txt -o salida4.txt < entrada4.txt
$ echo $?
0
$ cat salida4.txt
test1: 4 3.2e-11
```

1.4.10. Informe

El informe deberá incluir:

- Documentación relevante al diseño e implementación del programa.
- Documentación relevante a los algoritmos involucrados en la solución del trabajo.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje *C++* (en dos formatos, digital e impreso).
- Este enunciado

1.4.11. Fechas

La última fecha de entrega es el jueves 14 de junio.

Referencias

- (1) E. Gamma, R. Helm, R. Johnson J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st Ed., Addison-Wesley, 1994. Pags. 315-325.
- (2) Alan V. Oppenheim, Roland W. Schaffer. Discrete-Time Signal Processing, 2nd Ed., Prentice Hall, 1999. Cap. 9.3
- (3) T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. Introduction to Algorithms, McGraw-Hill Higher Education, 2nd ed., 2001. Cap. 30.2

2. Informe

2.1. Diseño e implementación

Para implementar un programa con los requisitos de diseño explicados en la sección 1.4.1 se estableció una nueva clase a modo de interfaz denominada "program" que normaliza los métodos requeridos para ejecutar el programa.

Considerando el feedback recibido en el *TP1* se realizó también una reorganización de las clases presentes en el archivo "fourier.cpp" separando cada una en sus correspondientes archivo de cabecera (.h) e implementación (.cpp) en línea con las buenas prácticas programación.

También se encapsuló a un grupo de funciones diversas tareas que se repetían a lo largo del código (ahora en archivo "utils.h") y se agregaron más comentarios a las que realizan operaciones más intrincadas para mejorar su lectura (como por ejemplo las que realizan operaciones de bits).

Así mismo, debido a la incorporación del preprocesamiento de regresiones debimos actualizar el flujo y el almacenamiento de resultados de nuestro programa como detallaremos a continuación.

Flujo del programa

1. Al invocar el ejecutable debemos leer la entrada de línea de comando
2. Debemos "parsear" (interpretar) opciones de entrada leídas
3. Se seleccionan "streams" de entrada, salida y regresión en función de las opciones leídas
4. Identificando el modo (interpretado de las opciones de la línea de comando), se inicializa la clase correspondiente con éste.
5. Si existe el parametro de regresion, se "marca" (flag) la instancia de clase strategy, para que reconozca que se debe procesar la regresion.
6. Invocando al método de programa ".compute()" se procede a:
 - a) leer el "stream" de entrada y almacenar los datos en una "lista entrada" llenando con ceros la misma al final hasta completar el tamaño de la potencia de 2 más cercana.
 - b) aplicar el algoritmo correspondiente al método seleccionado sobre la entrada (DFT, IDFT, FFT o IFFT) almacenando el resultado del mismo en una "lista salida" (con objeto de ser post-procesado)
 - c) si un archivo de regresiones fue provisto, computamos el error relativo entre dicha entrada y el resultado de nuestro algoritmo
 - d) escribimos la salida de nuestro algoritmo final (o resultados de test de regresion) al "stream" de salida seleccionado
7. Si no hay errores en alguno de los pasos anteriores, terminamos el programa con código 0.

2.1.1. Ubicación de código fuente

Para la realización de este trabajo optamos por emplear un repositorio acceso público y online. El mismo se encuentra alojado en <https://github.com/cristiandouce/tp2-algo2>.

2.1.2. Estructura del código fuente

Para desarrollar este programa se eligió la siguiente estructura de archivos y carpetas:

```

/
|- README.md    # Descripción de elementos del repositorio
|- main.cpp     # Punto de entrada del programa
|- vendor/      # Fuentes y headers provistos por la ctedra
|- src/         # Fuentes y headers desarrollados por los autores

```



```
| - bin/           # Salida de la compilación del código fuente: ejecutables
| - .vscode/       # Archivos auxiliares para del IDE: VSCode
```

2.1.3. Vendors / Librerías de la cátedra

Con objeto de poder concretar este trabajo práctico dentro de los tiempos de cursada, la cátedra ha provisto con algunas librerías con algunos de los problemas de implementación resueltos y que han sido explicados en clase. Estos son:

- `< cmdline.h > #cmdl()`: Clase de lectura de línea de comando. Recibe una tabla de opciones a leer por el programa a la hora de parsear.
- `< cmdline.h > #parse()`: Parseo de opciones de entrada. Guarda variables declaradas en el scope del archivo `main.cpp`. En este caso: los punteros a streams de entrada y salida, y el modo de algoritmo a ejecutar.
- `< complejo.h >`: Clase de complejos. Ofrece sobrecargas de operadores de suma y de lectura y escritura a streams.
- `< lista.h >`: Clase de lista. Ofrece implementación de listas doblemente enlazadas, no circulares, usando templates e iteradores.

Todas las fuentes de la cátedra han sido ubicados el directorio `vendor/` como se describió en la sección 2.1.2 y presentan modificaciones solo sobre los comentarios y estilo de código, pero no de declaraciones ni definiciones.

2.1.4. Fuentes de los autores

En base al requerimiento de la sección 1.4.1, se crearon una serie de clases para separar las interfaces de las implementaciones en los algoritmos de cálculos de DFT/FFT. Asimismo también se encapsularon ciertas funciones de uso habitual en el fichero “utils”. Las mismas se describen brevemente a continuación:

- `< fourier.h >`: Clase abstracta como interface para los algoritmos de calculo.
- `< dft.h >`: Clase que implementa los algoritmos de calculo de DFT por medio de la expresión literal.
- `< idft.h >`: Clase que implementa los algoritmos de calculo de IDFT por medio de la expresión literal.
- `< fft.h >`: Clase que implementa los algoritmos de calculo de FFT por medio de recursividad.
- `< ifft.h >`: Clase que implementa los algoritmos de calculo de IFFT por medio de recursividad.
- `< utils.h >`: Contiene funciones que encapsulan algoritmos utilizados asiduamente tales como el que extiende los vectores a potencias de 2.

Todas las fuentes de los autores han sido ubicados el directorio `src/` y son parte adjunta de este informe en la sección 2.8.

2.2. Compilación del programa

Para compilar el programa hay 2 opciones que se pueden utilizar de forma intercambiable.

Visual Studio Code

Junto con el código fuente se entrega un directorio `.vscode/` con una serie de archivos que ofrecen tareas de compilación para ser utilizadas con el **IDE** VSCode. Dentro de la lista de instrucciones se encuentran 2 posibles opciones.

La primera `build`, ofrece la compilación de los sources para sistemas de tipo `UNIX` empleando el compilador `g++` que debe ser instalado en el sistema. La segunda opción denominada `build_win10` ofrece generar el mismo resultado pero para sistemas operativos `Windows` y también requiere de la pre-instalación de la

herramienta de compilación `g++`.

Un tutorial sobre como usar el **IDE** y describe cómo configurar un Debugger y otras tareas se encuentra en **YouTube** con el titulo de VS Code: C++ Development With Visual Studio Code.

Makefile

La otra forma más estándar es utilizando la herramienta `make` para dirigir la compilación o generación de ejecutables automáticamente.

Como resultado de cualquier procedimiento utilizaddo un ejecutable será ubicado bajo el directorio `bin/` con el nombre `tp2` que podrá ejecutarse con la interfaz que es solicitada como requisito de este trabajo.

IMPORTANTE: Para los sistemas tipo UNIX solo es necesario tener instalado los utilitarios de `g++` para compilar el programa y así también correr los tests. Para el sistema operativo *Windows* se requiere de la instalación de *MinGW* disponible en <https://mingw.org> que provee de una serie de utilitarios minimos de tipo UNIX en Windows.

2.3. Análisis de complejidad de los algoritmos

En primera instancia vamos a denominar algunas definiciones básicas:

- n : Corresponde al número de puntos de una señal a ser procesada.
- N : Corresponde al tamaño de la siguiente potencia de 2 de la señal de entrada al programa. Es $N = 2^{\lceil \log_2(n) \rceil}$, donde $\lceil \cdot \rceil$ es la función matematica `ceil()`. Este es el valor real a ser procesado por el algoritmo. **Notar que en el peor caso $N = 2 \cdot n$.**
- d : Corresponde a la diferencia entre el tamaño de la señal de entrada n y su próxima potencia de 2 N , así $d = N - n$. **Notar que en el peor caso $d = n$.**
- m : Corresponde al número de señales a ser procesadas por archivo.
- $T(n)$: Implica el tiempo de procesamiento de un algoritmo para una entrada de tamaño n .
- $S(n)$: Implica el espacio utilizado de un algoritmo para una entrada de tamaño n .

Pasamos ahora a analizar los algoritmos *DFT* y *FFT*, sabiendo que sus correspondientes inversas son una variación de la original solo por un factor de conversión.

2.3.1. DFT

Recordando el algoritmo de la *DFT*

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \quad k = 0, \dots, N-1; \quad W_N = e^{-j(2\pi/N)} \quad (2)$$

En cuanto al costo temporal, podemos rápidamente inferir que para procesar un solo elemento de la salidad, tendremos que recorrer los n elementos de la entrada. Y siendo que el tamaño final del resultado de la *DFT* completada (bajo sub-índice k) es el mismo que el de la entrada, sabemos que realizaremos como mínimo un total de $n \cdot n$ iteraciones para poder construir la **Transformada Discreta de Fourier**.

Antes de poder concluir cuál es el resultado del costo temporal de nuestro algoritmo, procedemos a revisar su implementación en la sección 2.8.5. Algunos elementos a considerar son:

- Retorno rápido: Caso base de costo constante, $O(1)$
- Relleno de 0s a la derecha: Costo de procesamiento $O(\log(n))$ por el cálculo que averigua N y otro costo aditivo de $O(N - n) = O(d)$ por las operaciones que agregan los 0s al final de la lista.

- Operación del algoritmo: Costo de procesamiento de $O(N) \cdot O(N) = O(N^2)$
- Inserción del elemento procesado en una lista: Costo constante de $O(1)$ por como es la implementación del método `lista.insertar_despues()` en la sección 2.8.20. Esta operación se realiza un total de N veces dentro del loop del primer `for`.

Analizando rápidamente lo destacado más arriba podemos concluir la suma total del costo temporal de la *DFT* así implementada como la suma de:

$$T_{dft}(n, N, d) \in O(1) + O(\log(n)) + O(d) + O(N^2) + N \cdot O(1) \quad (3)$$

Pero como señalamos más arriba dentro de esta sección, en el peor de los casos tenemos que $N = 2 \cdot n$ y $d = n$. Por lo que la expresión final del costo temporal de la *DFT* termina siendo:

$$T_{dft}(n, N, d) \in O(1) + O(\log(n)) + O(n) + O(4 \cdot n^2) + 2 \cdot n \cdot O(1) \quad (4)$$

$$T_{dft}(n, N, d) \in O(\log(n)) + O(n) + 4 \cdot O(n^2) + 2 \cdot O(n) \quad (5)$$

$$T_{dft}(n, N, d) \in O(n^2) \quad (6)$$

$$T_{dft}(n) \in O(n^2) \quad (7)$$

En cuanto a la complejidad espacial, podemos simplemente observar de nuestro algoritmo que si la entrada es de tamaño n , el computo (y almacenamiento) será de un tamaño $N = 2 \cdot n$ en el peor de los casos. Así, sabiendo que mantenemos durante todo el procesamiento una *lista* de entrada y otra de salida, la complejidad espacial final es:

$$S_{dft}(n) \in O(n) \quad (8)$$

2.3.2. FFT

De forma análoga a la *DFT* recordamos primero el algoritmo de resolución de la *FFT* (como una implementación de la técnica **Divide & Conquer**):

$$X[k] = G[k] + W_N^{-kn} H[k] \quad k = 0, \dots, N-1; \quad W_N = e^{-j(2\pi/N)} \quad (9)$$

Donde $G(k)$ y $H(k)$ son las *DFTs* de $N/2$ puntos de las secuencias $x_0; x_2; \dots; x_{N-2}$ y $x_1; x_3; \dots; x_{N-1}$, es decir, los puntos con subíndices pares e impares respectivamente, donde se puede acceder a los vectores G y H de forma modular: $G(N/2) = G(0)$, $G(N/2 + 1) = G(1)$ y así sucesivamente.

Con una rápida inspección de este algoritmo podemos llegar al resultado de que en total, dada una entrada de tamaño n , llegaremos a realizar un total de $\log_2(n)$ llamadas recursivas, cada una de tamaño $\frac{n}{2}$. Considerando también la reconstrucción de la señal por sus partes par e impar, con un impacto de $O(n)$, encontramos que la *FFT* tiene un coste temporal de $O(n \cdot \log(n)) + O(n) \in O(n \cdot \log(n))$.

De igual manera que para la *DFT*, antes de concluir cuál es el resultado del costo temporal de nuestro algoritmo, procedemos a revisar su implementación en la sección 2.8.9. Algunos de los elementos a considerar son:

- Retorno rápido: Caso base de costo constante, $O(1)$
- Relleno de 0s a la derecha: Costo de procesamiento $O(\log(n))$ por el cálculo que averigua N y otro costo aditivo de $O(N - n) = O(d)$ por las operaciones que agregan los 0s al final de la lista.
- Operación del algoritmo: Costo de procesamiento de $O(N \cdot \log(N))$ como analizamos previamente que se analiza desde el primer llamado a `fft :: recursive_algorithm()` incluyendo el costo de partición $O(N)$ llamadas recursivas $2 \cdot O(\frac{N}{2})$ y reconstrucción $O(N)$.

Finalmente, sumando los costes de los componentes enumerados se llega a que:

$$T_{fft}(n, N, d) \in O(1) + O(\log(n)) + O(d) + O(N \cdot \log(N)) \quad (10)$$

$$T_{fft}(n, N, d) \in O(\log(n)) + O(n) + O(2 \cdot n \cdot \log(2 \cdot n)) \quad (11)$$

$$T_{fft}(n, N, d) \in O(\log(n)) + O(n) + 2 \cdot O(n \cdot (\log(2) + \log(n))) \quad (12)$$

$$T_{fft}(n, N, d) \in O(\log(n)) + O(n) + 2 \cdot O(n \cdot \log(2)) + 2 \cdot O(n \cdot \log(n)) \quad (13)$$

$$T_{fft}(n, N, d) \in O(\log(n)) + (2 \cdot \log(2) + 1) \cdot O(n) + 2 \cdot O(n \cdot \log(n)) \quad (14)$$

$$T_{fft}(n, N, d) \in O(n \cdot \log(n)) \quad (15)$$

$$T_{fft}(n) \in O(n \cdot \log(n)) \quad (16)$$

En cuanto a la complejidad espacial, observamos que, en cada llamada recursiva, guardamos estado para los dos arreglos auxiliares *odd* y *even*, cada uno de $n/2$ elementos. En definitiva, cada llamada usa memoria $O(n)$. El máximo consumo de memoria se va a dar cuando esten anidadas las $O(\log n)$ total de llamadas sobre las sucesivas porciones izquierdas del arreglo (los llamados de *odd* primero), de lo cual sigue que la complejidad espacial $S(n)$ viene dada por

$$S_{fft}(n) = n + \frac{n}{2} + \frac{n}{4} + \dots + 1 \quad (17)$$

Que en definitiva converge a $O(n \cdot \log(n))$, resultando entonces:

$$S_{fft}(n) \in O(n \cdot \log(n)) \quad (18)$$

2.4. Análisis de complejidad del programa completo

De igual manera al análisis de complejidad de los algoritmos, empezamos éste con las siguientes definiciones:

- n : Corresponde al número de puntos de una señal a ser procesada.
- N : Corresponde al tamaño de la siguiente potencia de 2 de la señal de entrada al programa. Es $N = 2^{\lceil \log_2(n) \rceil}$, donde $\lceil \cdot \rceil$ es la función matemática *ceil*(\cdot). Este es el valor real a ser procesado por el algoritmo. **Notar que en el peor caso $N = 2 \cdot n$.**
- d : Corresponde a la diferencia entre el tamaño de la señal de entrada n y su próxima potencia de 2 N , así $d = N - n$. **Notar que en el peor caso $d = n$.**
- m : Corresponde al número de señales a ser procesadas por archivo.
- $T(n)$: Implica el tiempo de procesamiento de un algoritmo para una entrada de tamaño n .
- $S(n)$: Implica el espacio utilizado de un algoritmo para una entrada de tamaño n .

Pasamos ahora a analizar los procesos de *entrada*, *salida* y el *conjunto* del programa.

2.4.1. Procesamiento de lectura y almacenamiento

Para analizar los costos relacionados con el procesamiento de la entrada de la señal vamos a realizar la siguiente hipótesis: **De las m señales en nuestro archivo de entrada, denominamos n a aquella de mayor tamaño.**

Con esta hipótesis que solo extiende los comentarios de la sección anterior, vamos a proceder con el análisis del procesamiento de la entrada.

Pondremos especial atención a 2 funciones particulares: `1.ft :: read_input_line()` de la clase *ft* en `src/fourier.cpp` 2.8.3 y `2.read_input_stream_line()` de nuestra librería `src/utls.cpp` de la sección 2.8.13.

Consideremos primero que la función *ft* :: `read_input_line()` se asegura que la lista en la que se almacenará la primer línea de entrada se encuentre vacía. Para ello invoca *lista* :: `clear()` que realiza un recorrido en

tiempo $O(n)$ para liberar todos los elementos previos que pueda tener almacenados. **Recordar la hipótesis de n como peor/mayor tamaño de señal en el archivo.**

Luego de ello, se invoca la función *read_input_stream_line()* que lo que hace es leer la primera línea del stream de entrada, ingresando cada elemento de tipo *complejo* encontrado en la lista *input_* que se asume previamente liberada. Esta tarea, de nuevo considerando n el peor tamaño de señal dentro del archivo de lectura, se realiza en un tiempo $O(n)$.

En consecuencia de lo analizado hasta ahora, tenemos que el costo temporal del procesamiento de la lectura y almacenamiento de 1 señal es de:

$$T_{lectura}(n) \in O(n) \quad (19)$$

En cuanto a la complejidad espacial:

$$S_{lectura}(n) \in O(n) \quad (20)$$

2.4.2. Procesamiento de escritura

De igual manera que en el procesamiento de lectura vamos a valernos de la hipótesis del n más grande dentro del archivo de señales por línea para definir nuestro análisis.

Pondremos especial atención a 2 funciones particulares: 1. *ft* :: *write_output_line()* de la clase *ft* en *src/fourier.cpp* 2.8.3 y 2. *write_output_stream_line()* de nuestra librería *src/utls.cpp* de la sección 2.8.13.

De igual manera que para la lectura, en este caso la escritura se encarga de enviar cada dato almacenado en la lista *output_* post-procesado por nuestro algoritmos, y luego asegurarse de dejar limpia dicha lista para poder ser utilizada para el procesamiento de la siguiente señal. En consecuencia, y al igual que para la lectura, la complejidad del procesamiento de escritura es:

$$T_{escritura}(n) \in O(n) \quad (21)$$

$$S_{escritura}(n) \in O(n) \quad (22)$$

2.4.3. Procesamiento conjunto del programa

Teniendo en cuenta todo lo enunciado hasta el momento en el análisis de complejidad de los algoritmos y los procedimientos de lectura y escritura del programa, podemos concluir que los tiempos conjuntos para cada rama (algoritmo) serán:

Complejidad temporal

$$T_{dft}^{total}(n) = T_{lectura}(n) + T_{dft}(n) + T_{escritura}(n) \quad (23)$$

$$T_{dft}^{total}(n) \in O(n) + O(n^2) + O(n) \quad (24)$$

$$T_{dft}^{total}(n) \in O(n^2) \quad (25)$$

$$T_{fft}^{total}(n) = T_{lectura}(n) + T_{fft}(n) + T_{escritura}(n) \quad (26)$$

$$T_{fft}^{total}(n) \in O(n) + O(n \cdot \log(n)) + O(n) \quad (27)$$

$$T_{fft}^{total}(n) \in O(n \cdot \log(n)) \quad (28)$$

Complejidad espacial

$$S_{dft}^{total}(n) = S_{lectura}(n) + S_{dft}(n) + S_{escritura}(n) \quad (29)$$

$$S_{dft}^{total}(n) \in O(n) + O(n) + O(n) \quad (30)$$

$$S_{dft}^{total}(n) \in O(n) \quad (31)$$

$$S_{fft}^{total}(n) = S_{lectura}(n) + S_{fft}(n) + S_{escritura}(n) \quad (32)$$

$$S_{fft}^{total}(n) \in O(n) + O(n \cdot \log(n)) + O(n) \quad (33)$$

$$S_{fft}^{total}(n) \in O(n \cdot \log(n)) \quad (34)$$

Y considerando las m señales del archivo (recordando también que entre señal y señal, las listas de entrada y salida se renuevan):

$$T_{dft}^{total}(n, m) \in m \cdot O(n^2) \quad (35)$$

$$S_{dft}^{total}(n, m) \in O(n) \quad (36)$$

$$T_{fft}^{total}(n, m) \in m \cdot O(n \cdot \log(n)) \quad (37)$$

$$S_{fft}^{total}(n, m) \in O(n \cdot \log(n)) \quad (38)$$

2.5. Pruebas de performance

Para nuestros test de performance vamos a definir 3 archivos con una cantidad de puntos equivalente, solo que distribuidos de distinta manera entre las señales de entrada. Con esto realizaremos algunas observaciones cualitativas y conclusiones cuantitativas respecto a los resultados obtenidos de nuestro análisis de complejidad en la sección 2.4.3.

Los archivos que empleamos son:

- *test/files/test_1_signals_10000_points.txt*: Archivo que contiene una señal con 10k puntos. Son 10k puntos en total.
- *test/files/test_10_signals_1000_points.txt*: Archivo que contiene 10 señales de 1k puntos cada una. Son 10k puntos en total.
- *test/files/test_100_signals_100_points.txt*: Archivo que contiene 100 señales de 100 puntos cada una. Son 10k puntos en total.

Para poder correr estos tests generamos una entrada en nuestro *Makefile* que ejecuta un **shell script** denominado *./test/tests.sh* adjunto en la sección ?? Lo que hace dicho shell script es correr cada uno de los comandos $x100$ veces para la *FFT* y $x10$ veces para el modo *DFT* para poder obtener resultados numéricos significativos, ya que de lo contrario deberíamos trabajar con magnitudes del orden de los mili o nano segundos.

Algo importante a resaltar es que para las pruebas de performance utilizamos la herramienta **built-in** de sistemas UNIX denominada *time*, que devuelve como resultado 3 valores numéricos a destacar:

- **real**: El tiempo real invertido en ejecutar el proceso de principio a fin, como si lo midiera un humano con un cronómetro.
- **user**: El tiempo acumulado gastado por todas las CPU durante el procesamiento del comando.
- **system**: El tiempo acumulado por todas las CPU durante las tareas relacionadas con el sistema, como asignación de memoria, lectura / escritura a archivos.

Con esta información vamos a intentar extrapolar los tiempos de ejecución del algoritmo, de escritura y lectura de archivos.

Empezamos ahora por intentar predecir los resultados de correr los tests para la *FFT* y luego deduciremos lo mismo para la *DFT*.

Partiendo desde la base de que los archivos son estáticos y no cambia entre múltiples corridas. Considerando también conocida y constante la cantidad de puntos para todos los archivos, nuestro objetivo será

utilizar la definición de O grande para encontrar un valor de referencia que sea comparable entre las diferentes corridas.

Así recopilamos primero los siguientes parámetros de cada uno de los archivos de corrida:

100 señales de 100 puntos

$$m_{100} = 100 \quad n_{100} = 100 \quad N_{100} = 2^7 = 128 \quad (39)$$

10 señales de 1000 puntos

$$m_{1000} = 10 \quad n_{1000} = 1000 \quad N_{1000} = 2^{10} = 1024 \quad (40)$$

1 señal de 10000 puntos

$$m_{10000} = 1 \quad n_{10000} = 10000 \quad N_{10000} = 2^{14} = 16384 \quad (41)$$

Y las relaciones que buscamos entre las distintas corridas serán:

$$test_{10000-100}^{fft} = \frac{m_{10000}}{m_{1000}} \cdot \frac{T_{fft}(N_{10000})}{T_{fft}(N_{1000})} \quad (42)$$

$$test_{1000-100}^{fft} = \frac{m_{1000}}{m_{100}} \cdot \frac{T_{fft}(N_{1000})}{T_{fft}(N_{100})} \quad (43)$$

$$test_{10000-100}^{fft} = \frac{m_{10000}}{m_{100}} \cdot \frac{T_{fft}(N_{10000})}{T_{fft}(N_{100})} \quad (44)$$

Tomando la definición de notación de la O grande, sabiendo que si existe una diferencia relativa entre corridas, a la misma entrada, es de una constante que se cancela con el cociente:

$$test_{10000-1000}^{fft} = \frac{m_{10000}}{m_{1000}} \cdot \frac{N_{10000} \cdot \log(N_{10000})}{N_{1000} \cdot \log(N_{1000})} = 2,24 \quad (45)$$

$$test_{1000-100}^{fft} = \frac{m_{1000}}{m_{100}} \cdot \frac{N_{1000} \cdot \log(N_{1000})}{N_{100} \cdot \log(N_{100})} = 1,14 \quad (46)$$

$$test_{10000-100}^{fft} = \frac{m_{10000}}{m_{100}} \cdot \frac{N_{10000} \cdot \log(N_{10000})}{N_{100} \cdot \log(N_{100})} = 2,56 \quad (47)$$

Lo que podemos inferir de este análisis es que entre los resultados de ejecución en tiempo de los tests de performance deberíamos notar una gran distinción entre los tiempos de 100 (o 1k) puntos por señal con la de 10kpuntos por señal. Mientras que en el salto entre 100 y 1k puntos por señal, el tiempo es casi indistinguible ya que es del orden de 1,14.

Repetimos lo mismo ahora para el mismo análisis para la DFT :

$$test_{10000-1000}^{dft} = \frac{m_{10000}}{m_{1000}} \cdot \frac{N_{10000}^2}{N_{1000}^2} = 25,6 \quad (48)$$

$$test_{1000-100}^{dft} = \frac{m_{1000}}{m_{100}} \cdot \frac{N_{1000}^2}{N_{100}^2} = 6,4 \quad (49)$$

$$test_{10000-100}^{dft} = \frac{m_{10000}}{m_{100}} \cdot \frac{N_{10000}^2}{N_{100}^2} = 163,8 \quad (50)$$

Analizando los resultados de las corridas disponibles en la sección ?? *test/results/perf - total.txt* podemos generar los siguientes resultados relativos:

$$test_{10000-1000}^{fft} = \frac{15,088}{7,329} = 2,06 \cong 2,24 \quad (51)$$

$$test_{1000-100}^{fft} = \frac{7,329}{7,216} = 1,014 \cong 1,14 \quad (52)$$

$$test_{10000-100}^{fft} = \frac{15,088}{7,216} = 2,09 \cong 2,56 \quad (53)$$

$$test_{10000-1000}^{dft} = \frac{284,991}{11,419} = 24,96 \cong 25,6 \quad (54)$$

$$test_{1000-100}^{dft} = \frac{11,419}{2,051} = 5,57 \cong 6,4 \quad (55)$$

$$test_{10000-100}^{dft} = \frac{284,991}{2,051} = 138,95 \cong 163,8 \quad (56)$$

En definitiva, los resultados obtenidos por nuestros tests de performance son **congruentes** con los predichos por nuestro análisis. El hecho de que no sean idénticos tiene mucho sentido, ya que como mencionamos antes, por la definición de $O(n)$, hablamos de una tendencia a partir de algún $n \geq n_0$. Pero lo predicho nos permite determinar algún orden de magnitud entre los resultados obtenidos.

Ahora terminando, intentaremos determinar los tiempos de lectura y procesamiento de la entrada en forma relativa, los de las corridas del algoritmo y los de escritura. Utilizando nuevamente los resultados de tipo **real** de los archivos *perf - total.txt*, *perf - lectura.txt* y *perf - lectura - y - alg.txt* en los que intencionalmente se han deshabilitado las funciones del algoritmo y de escritura de resultados para poder determinar tiempos **relativos** entre los 3 componentes principales del programa podemos observar lo siguiente:

$$time_{total}^{fft10k} = 15,088s \quad (57)$$

$$time_{lectura}^{fft10k} = 1,750s \approx 11,6\% \quad (58)$$

$$time_{lectura+algoritmo}^{fft10k} = 13,770s \quad (59)$$

$$time_{escritura}^{fft10k} = time_{total}^{fft10k} - time_{lectura+algoritmo}^{fft10k} \quad (60)$$

$$= 1,318s \approx 8,74\% \quad (61)$$

$$time_{algoritmo}^{fft10k} = time_{lectura+algoritmo}^{fft10k} - time_{lectura}^{fft10k} \quad (62)$$

$$= 12,02s \approx 79,7\% \quad (63)$$

El mismo ejemplo se puede repetir para todos los valores de de corridas, de 1k puntos por 10 señales y 100 puntos por 100 señales. En particular mostraremos ahora el de la *DFT*:

$$time_{total}^{dft10k} = 284,991s \quad (64)$$

$$time_{lectura}^{dft10k} = 0,120s \approx 0,04\% \quad (65)$$

$$time_{lectura+algoritmo}^{dft10k} = 270,603s \quad (66)$$

$$time_{escritura}^{dft10k} = time_{total}^{dft10k} - time_{lectura+algoritmo}^{dft10k} \quad (67)$$

$$= 14,307 \approx 5,02\% \quad (68)$$

$$time_{algoritmo}^{dft10k} = time_{lectura+algoritmo}^{dft10k} - time_{lectura}^{dft10k} \quad (69)$$

$$= 270,483s \approx 94,9\% \quad (70)$$

Podemos ver que para los calculos de tiempo de escritura y son más cercanos en comparación cuando ejecutamos el algoritmo de la *FFT* que en la *DFT*, resulta concluir que al tomar un tiempo tan significativo para procesar el algoritmo, ésta hace infinitesimal el tiempo de operación de los otros procesos del programa.

Si repetimos para el experimento de 100 señales de 100 puntos obtenemos algo más razonable:

$$time_{total}^{dft100} = 2,051s \quad (71)$$

$$time_{lectura}^{dft100} = 0,106s \approx 5,17\% \quad (72)$$

$$time_{lectura+algoritmo}^{dft100} = 1,784s \quad (73)$$

$$time_{escritura}^{dft100} = time_{total}^{dft100} - time_{lectura+algoritmo}^{dft100} \quad (74)$$

$$= 0,267s \approx 13,01\% \quad (75)$$

$$time_{algoritmo}^{dft100} = time_{lectura+algoritmo}^{dft100} - time_{lectura}^{dft100} \quad (76)$$

$$= 270,483s \approx 94,9\% \quad (77)$$

2.6. Troubleshooting

Para cualquier lenguaje de programación es importante poder detectar errores antes y después de la compilación o ejecución de nuestros programas. Para ello incorporamos en nuestro análisis el uso de 2 herramientas básicas al programar:

1. Memory Leak Detector: Para detectar malos usos de memoria en nuestro programa.
2. Static Analyzer: Para poder detectar errores propios de nuestro código como **tipos** o **unused variables** que nos lleven a que nuestro programa o bien no compile o lo haga de forma poco eficiente.

2.6.1. Memory Leak Detection

Como herramienta de **Memory Leal Detection** empleamos el utilitario *valgrind* en su modo *memcheck*.

Cabe mencionar que debido a que el equipo de desarrollo de dicha herramienta prioriza los sistemas operativos basados en Linux, no nos fue posible emplear la herramienta en sistemas operativos de tipo OSX o Windows.

Para su evaluación en los mencionados sistemas operativos nos fue necesario o bien montar una máquina virtual con sistema operativo Ubuntu usando *VirtualBox*. O bien, de modo más avanzado, levantar *Docker* containers que corran la misma.

En nuestro caso optamos por la primera, pues ya teníamos una máquina virtual instalada con Ubuntu 14.

Para correr los tests de Memory Leak basta solo (cumpliendo los requisitos de Sistema Operativo) correr el comando *maketest - valgrind* que ejecuta lo siguiente:

```
test-valgrind: bin/tp2
# memcheck is default --tool
valgrind --leak-check=yes $(OUTFILE) -i ./test/files/test_1-signals-10000-points
.txt -o ./test/files/_garbage.txt
valgrind --leak-check=yes $(OUTFILE) -i ./test/files/test_10-signals-1000-points
.txt -o ./test/files/_garbage.txt
valgrind --leak-check=yes $(OUTFILE) -i ./test/files/test_100-signals-100-points
.txt -o ./test/files/_garbage.txt
```

Como puede observarse decidimos usar los mismos archivos de prueba que en nuestros Test de Performance de la sección ??, ya que los mismos proveen de 2 escenarios básicos: 1. Una señal grande, 2. Varisa señales.

En el caso de detectar algún Memory Leak, cada una e las entradas anteriores produciría una salida en el siguiente formato:

```

$ make test-valgrind
==3632== Memcheck, a memory error detector
==3632== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3632== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3632== Command: ./bin/tp2
==3632==

ptr = [Linux]

ptr = [ainux]
==3632==
==3632== HEAP SUMMARY:
==3632==      in use at exit: 10 bytes in 1 blocks
==3632==    total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==3632==
==3632== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3632==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.
    so)
==3632==    by 0x8048428: main (main.c:7)
==3632==
==3632== LEAK SUMMARY:
==3632==    definitely lost: 10 bytes in 1 blocks
==3632==    indirectly lost: 0 bytes in 0 blocks
==3632==    possibly lost: 0 bytes in 0 blocks
==3632==    still reachable: 0 bytes in 0 blocks
==3632==    suppressed: 0 bytes in 0 blocks
==3632==
==3632== For counts of detected and suppressed errors, rerun with: -v
==3632== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

```

Es necesario mencionar que para el caso de este trabajo práctico, luego de varios esfuerzos por hacer funcionar la herramienta, una vez logrado, no pudimos hacer saltar algún Memory Leak observable como para arreglar.

2.6.2. Static Analyzer

Para la realización de este trabajo práctico, como bien explicamos ya en la sección 2.2 utilizamos el **IDE Visual Studio Code** que ya viene integrado con una herramienta de análisis estático del código. La descripción de la misma se encuentra en:

- <https://github.com/Microsoft/vscode-cpptools/blob/master/Documentation/LanguageServer/IntelliSense%20engine.md>.

Durante el desarrollo de este trabajo práctico estos fueron los errores más comunes que se nos alertó por la herramienta:

- Inexistencia de sobrecarga para la forma que se utiliza cierto método.
- Inexistencia de declaración de algún método en el encabezado *.h.
- Variable declarada pero nunca empleada
- Variable no definida

La detección de estos errores previo paso de compilación es de vital importancia para un programador pues le permite ahorrar tiempo ya que la etapa de compilación puede tomar (para programas de tamaño mucho mayor) un tiempo en el orden de los minutos.

2.7. Análisis y conclusiones de resultados

Durante la evaluación del programa compilado logramos recopilar la siguiente información:

1. Podemos observar cuantitativamente la eficiencia entre los resultados de los algoritmos de *FFT* frente a los *DFT*, siendo los resultados de aproximadamente 0,52 % la relación entre *FFT* contra *DFT* en tiempos de ejecución para una misma señal.
2. Mediante el análisis de Complejidad Temporal del programa completo es posible estimar el tiempo relativo de corridas del programa ante 2 entradas distintas. No obstante, estimación exacta de dicho tiempo depende de muchos otros factores que son más difíciles de predecir.
3. Para poder determinar con mejor precisión los tiempos involucrados en lectura, procesamiento de algoritmo y escritura a salida estándar o archivo, lo ideal sería desarrollar tests unitarios que pudieran realizar dichas operaciones de forma independiente, y luego predecir el comportamiento conjunto (como bien hicimos al principio de la sección 2.5 de Performance) y no a la inversa.
4. La distribución de los puntos de señal al programa es muy importante para su performance. Como ya vimos en la sección 2.5 cuando en el peor de los casos la señal se aleja del siguiente múltiplo de 2 en $n - 1$, el cálculo se hace más impredecible y con mayor error. Aquí es donde entran en juego distintas técnicas de ventaneo de señales por tramos de potencia de 2 que permitan analizar mejor el espectro de la señal deseada.
5. Pudimos verificar también mediante el mecanismo regresivo que el error cometido entre señales por DFT y FFT siempre menos a $1e - 3$ siendo el error más grande encontrado luego de varias iteraciones y pruebas del orden de $1e - 7$

2.8. Código fuente

2.8.1. main.cpp

```

1  /**
2   * Name : main.cpp
3   * Authors : Dreiling, Augusto <augusto.dreiling@gmail.com>
4   * : Douce Suarez, Cristian G. <cristiandouce@yahoo.com.ar>
5   * Version : 1.0.0
6   * License : MIT
7   * Description : Linea de comando que implementa FFT/IFFT/DFT/IDFT
8   * sobre senales de entrada de variable real o compleja.
9   */
10
11 #include <fstream>
12 #include <iomanip>
13 #include <iostream>
14 #include <sstream>
15 #include <cstdlib>
16
17 #include <string>
18 #include <algorithm>
19
20 #include "vendor/cmdline.h"
21 #include "vendor/complejo.h"
22 #include "src/fourier.h"
23 #include "src/dft.h"
24 #include "src/idft.h"
25 #include "src/fft.h"
26 #include "src/iff.h"
27
28 using namespace std;
29
30 static void opt_input(string const &);
31 static void opt_output(string const &);
32 static void opt_regression(string const &);
33 static void opt_method(string const &);
34 static void opt_error(string const &);
35 static void opt_help(string const &);
36

```

```

37 /**
38  * Tabla de opciones de linea de comando. El formato de la tabla
39  * consta de un elemento por cada opcion a definir. A su vez, en
40  * cada entrada de la tabla tendremos:
41  *
42  * o La primera columna indica si la opcion lleva (1) o no (0) un
43  * argumento adicional.
44  *
45  * o La segunda columna representa el nombre corto de la opcion.
46  *
47  * o La tercera columna determina el nombre largo de la opcion.
48  *
49  * o La cuarta columna contiene el valor por defecto a asignarle
50  * a esta opcion en caso que no esta explicitamente presente
51  * en la linea de comandos del programa. Si la opcion no tiene
52  * argumento (primera columna nula), todo esto no tiene efecto.
53  *
54  * o La quinta columna apunta al metodo de parseo de la opcion,
55  * cuyo prototipo debe ser siempre void (*m)(string const &arg);
56  *
57  * o La ultima columna sirve para especificar el comportamiento a
58  * adoptar en el momento de procesar esta opcion: cuando la
59  * opcion es obligatoria, debera activarse OPT_MANDATORY.
60  *
61  * Ademias, la ultima entrada de la tabla debe contener todos sus
62  * elementos nulos, para indicar el final de la misma.
63  */
64
65 /***** Elementos globales *****/
66 static option_t options[] = {
67     { 1, "i", "input", "-", opt_input, OPT_DEFAULT },
68     { 1, "o", "output", "-", opt_output, OPT_DEFAULT },
69     { 1, "r", "regression", NULL, opt_regression, OPT_DEFAULT },
70     { 1, "e", "error", "1e-3", opt_error, OPT_DEFAULT },
71     { 1, "m", "method", "FFT", opt_method, OPT_DEFAULT },
72     { 0, "h", "help", NULL, opt_help, OPT_DEFAULT },
73     { 0, },
74 };
75
76 enum Methods { DFT, IDFT, FFT, IFFT };
77
78 static Methods method;
79 static istream *iss = 0; // Input Stream (clase para manejo de los flujos de entrada)
80 static ostream *oss = 0; // Output Stream (clase para manejo de los flujos de salida)
81 static istream *rss = 0; // Regression Stream (clase para manejo de los flujos de entrada)
82 static fstream ifs; // Input File Stream (derivada de la clase ifstream que deriva de istream para el manejo de
    ↪ archivos)
83 static fstream ofs; // Output File Stream (derivada de la clase ofstream que deriva de ostream para el manejo de
    ↪ archivos)
84 static fstream rfs; // Regression File Stream (derivada de la clase ifstream que deriva de ostream para el manejo
    ↪ de archivos)
85 static double error;
86
87
88 /*****
89
90 static void
91 opt_input(string const &arg) {
92     // Si el nombre del archivos es "-", usaremos la entrada
93     // estandar. De lo contrario, abrimos un archivo en modo
94     // de lectura.
95     if (arg == "-") {
96         // Establezco la entrada estandar cin como flujo de entrada

```

```

97     iss = &cin;
98 } else {
99     // c_str(): Returns a pointer to an array that contains a null-terminated
100    // sequence of characters (i.e., a C-string) representing
101    // the current value of the string object.
102    ifs.open(arg.c_str(), ios::in);
103    iss = &ifs;
104 }
105
106 // Verificamos que el stream este OK.
107 //
108 if (!iss->good()) {
109     cerr << "cannot open " << arg << "." << endl;
110     // EXIT: Terminacion del programa en su totalidad
111     exit(1);
112 }
113 }
114
115 static void
116 opt_output(string const &arg) {
117     // Si el nombre del archivos es "-", usaremos la salida
118     // estandar. De lo contrario, abrimos un archivo en modo
119     // de escritura.
120     if (arg == "-") {
121         // Establezco la salida estandar cout como flujo de salida
122         oss = &cout;
123     } else {
124         ofs.open(arg.c_str(), ios::out);
125         oss = &ofs;
126     }
127
128     // Verificamos que el stream este OK.
129     //
130     if (!oss->good()) {
131         cerr << "cannot open "
132             << arg
133             << "."
134             << endl;
135         // EXIT: Terminacion del programa en su totalidad
136         exit(1);
137     }
138 }
139
140 static void
141 opt_regression(string const &arg) {
142     if (!arg.empty()) {
143         // c_str(): Returns a pointer to an array that contains a null-terminated
144         // sequence of characters (i.e., a C-string) representing
145         // the current value of the string object.
146         rfs.open(arg.c_str(), ios::in);
147         rss = &rfs;
148     }
149
150     // Verificamos que el stream este OK.
151     //
152     if (!rss->good()) {
153         cerr << "cannot open " << arg << "." << endl;
154         // EXIT: Terminacion del programa en su totalidad
155         exit(1);
156     }
157 }
158
159 static void

```

```

160 opt_method(string const &arg) {
161     // Intentamos extraer el metodo de la linea de comandos (DFT, IDFT, FFT o IFFT).
162     //
163     if (arg == "DFT") {
164         // Establezco metodo como DFT
165         method = DFT;
166     } else if (arg == "IDFT") {
167         // Establezco metodo como IDFT
168         method = IDFT;
169     } else if (arg == "FFT") {
170         // Establezco metodo como IDFT
171         method = FFT;
172     } else if (arg == "IFFT") {
173         // Establezco metodo como IDFT
174         method = IFFT;
175     } else {
176         cerr << "La opcion 'method' provista es invalida."
177         << endl;
178         // EXIT: Terminacion del programa en su totalidad
179         exit(1);
180     }
181 }
182 static void
183 opt_error(string const &arg) {
184     error = atof(arg.c_str());
185 }
186
187 static void
188 opt_help(string const &arg) {
189     cout << "tp2 [-m method] [-i file] [-o file] [-r file] [-e error]"
190     << endl;
191     exit(0);
192 }
193
194 int
195 main(int argc, char * const argv[]) {
196     // Objeto con parametro tipo option_t (struct) declarado globalmente. Ver linea 51 main.cc
197     // Metodo de parseo de la clase cmdline
198     cmdline cmdl(options);
199     cmdl.parse(argc, argv);
200
201     ft *myft = 0;
202
203     if (method == DFT) {
204         myft = new dft(iss, oss);
205     } else if (method == IDFT) {
206         myft = new idft(iss, oss);
207     } else if (method == IFFT) {
208         myft = new ifft(iss, oss);
209     } else {
210         myft = new fft(iss, oss);
211     }
212
213     // flaggeo el ft para que calcule la regresion
214     if (rss != 0) {
215         myft->regression(rss, error);
216     }
217
218     // computar el resultado
219     myft->compute();
220
221     // obtener el codigo de la operacion
222     const int code = myft->code();

```

```

223
224 // destruir instancia
225 delete myft;
226
227 // retornar codigo del programa
228 return code;
229 }

```

2.8.2. src/fourier.h

```

1  #ifndef _FOURIER_H_INCLUDED_
2  #define _FOURIER_H_INCLUDED_
3
4  #include <fstream>
5  #include <sstream>
6  #include <cstdlib>
7  #include <cmath>
8  #include <complex>
9  #include <string>
10
11 #include "../vendor/lista.h"
12 #include "../vendor/complexo.h"
13
14 #include "../program.h"
15 #include "../src/utls.h"
16
17 using namespace std;
18
19 class ft : public program {
20
21     protected:
22         lista<complejo> input_;
23         lista<complejo> output_;
24         istream *is_;
25         ostream *os_;
26
27         // para la regresion
28         istream *rs_;
29         double rerr_;
30         int regrN_;
31
32         /**
33          * @brief lee una linea del stream de entrada
34          * y la guarda en input_ para ser procesada
35          * al llamar run_algoritmo()
36          */
37         void read_input_line();
38         // void read_input_stream_line(istream *, lista<complejo> &);
39
40         /**
41          * @brief Escribe una linea en el stream de output
42          * lo almacenado en el arreglo output_
43          */
44         void write_output_line();
45
46         double get_norm();
47         double get_norm(double const &);
48
49         complejo get_exp_complejo();
50
51         /**
52          * @brief Invoca el algoritmo de regression

```

```

53     *   para una linea del vector de entrada.
54     *
55     */
56     void run_regression();
57
58     /**
59     * @brief Calcula laa regresion entre 2 listas de complejos
60     *   escribiendo su resultado a os_, y afectando el
61     *   codigo del programa para ser retornado
62     *
63     */
64     void calculate_regression(lista<complejo> &, lista<complejo> &);
65
66     virtual bool inverse() = 0;
67
68     virtual void run_algorithm() = 0;
69
70     public:
71         ft();
72
73         ft(istream *is);
74
75         ft(ostream *os);
76
77         ft(istream *is, ostream *os);
78
79     virtual ~ft(){};
80
81
82     /**
83     * @brief Flaggea el la instancia para correr
84     *   la regression como salida del compute()
85     */
86     void regression(istream *rs, double const &);
87
88     /**
89     * @brief Inicializa el procesamiento del algoritmo
90     *   seleccionado.
91     *
92     */
93     void compute();
94 };
95
96 #endif

```

2.8.3. src/fourier.cpp

```

1  #include <iomanip>
2  #include <iostream>
3  #include <cmath>
4
5  #include "../fourier.h"
6  #include "../utils.h"
7
8  using namespace std;
9
10 ft::ft(): is_(&cin), os_(&cout), rs_(0), rerr_(0), regrN_(1) { }
11
12 ft::ft(istream *is): is_(is), os_(&cout), rs_(0), rerr_(0), regrN_(1) { }
13
14 ft::ft(ostream *os): is_(&cin), os_(&cout), rs_(0), rerr_(0), regrN_(1) { }
15

```



```

16 ft::ft(istream *is, ostream *os): is_(is), os_(os), rs_(0), rerr_(0), regrN_(1) { }
17
18 void ft::read_input_line() {
19     // nos aseguramos que el input_ lista
20     // este siempre vaco antes de empezar.
21     input_.clear();
22
23     // leemos desde la entrada seleccionada del programa
24     read_input_stream_line(is_, input_);
25 }
26
27 void ft::write_output_line() {
28     // NOTE: no escribo desde output_ cuando
29     // estoy en modo regression
30     if (rs_ != 0) {
31         return;
32     }
33
34     // escribimos a la salida del programa seleccionada
35     write_output_stream_line(os_, output_);
36
37     // nos aseguramos que el vector de salida
38     // este siempre vacio al terminar
39     output_.clear();
40 }
41
42 double
43 ft::get_norm () {
44     double N = input_.tamano();
45     return get_norm(N);
46 }
47
48 double
49 ft::get_norm(double const &N) {
50     return inverse() && (N > 0) ? 1/N : 1;
51 }
52
53 complejo
54 ft::get_exp_complejo() {
55     return inverse() ? complejo(0, -1) : complejo(0, 1);
56 }
57
58 void
59 ft::regression(istream *rs, double const &rerr) {
60     rs_ = rs;
61     rerr_ = rerr;
62 }
63
64 void ft::run_regression() {
65     // salgo de la ejecucion si la regresion
66     // no fue inicializada correctamente
67     if (rs_ == 0) {
68         return;
69     }
70
71     // copio y libero el arreglo de salida
72     lista<complejo> &Xk = output_;
73
74     // retorno rapido si Xk esta vacio
75     if (Xk.tamano() == 0) {
76         cout << endl;
77         return;
78     }

```

```

79
80 // leo la primer linea desde el stream de regresiones
81 lista<complejo> Rk;
82 read_input_stream_line(rs_, Rk);
83
84 // retorno rapido si Rk esta vacio
85 if (Rk.tamano() == 0) {
86     cout << endl;
87     return;
88 }
89
90 // completo los arreglos con 0s para poder calcular la regresion
91 int maxSize = max(Rk.tamano(), Xk.tamano());
92 unsigned int zerosToRk = maxSize - Rk.tamano();
93 unsigned int zerosToXk = maxSize - Xk.tamano();
94
95 zero_pad(Rk, zerosToRk);
96 zero_pad(Xk, zerosToXk);
97
98 // proceso la regresion y escribo a os_ directo.
99 // el write_output_line tiene una regla de escape
100 // para el escenario de regression
101 calculate_regression(Rk, Xk);
102
103 // Me aseguro de limpiar las variables al terminar de
104 // procesar la linea
105 Rk.clear();
106 Xk.clear();
107 }
108
109 void
110 ft::calculate_regression(lista<complejo> &X, lista<complejo> &R) {
111     if (X.tamano() != R.tamano()) {
112         cout << "Error interno" << endl;
113         exit(-1);
114     }
115
116     // En
117     if (X.tamano() == 0) {
118         *os_ << endl;
119         return;
120     }
121
122     lista<complejo>::iterador Xit = X.primer();
123     lista<complejo>::iterador Rit = R.primer();
124
125     double num;
126     double den;
127
128     while (!Xit.extremo()) {
129         complejo aux = Xit.dato() - Rit.dato();
130
131         // sumo en el numerador producto(aux, aux*)
132         num += aux.abs2();
133
134         // sumo en el denominador
135         // producto(R[k], R[k]*) = abs(R[x])^2
136         den += Rit.dato().abs2();
137
138         // avanzo ambos iteradores para el siguiente k
139         Xit.avanzar();
140         Rit.avanzar();
141     }

```

```

142
143 // calculo finalmente el error
144 double Ex = sqrt(num / den);
145
146 // evaluo si esta ok?
147 bool ok = Ex < rerr.;
148
149 // escribo a la salida del programa
150 *os_ << "test " << regrN_ << " : "
151     << (ok ? " ok " : " error ")
152     << X.tamano() << " "
153     << Ex
154     << endl;
155
156 // incremento el flag de cuantos tests se corrieron
157 regrN_++;
158
159 // marco el programa con error solo si
160 if (!ok) {
161     code(-1);
162 }
163 }
164
165 void
166 ft::compute() {
167     while(!is_-->eof()) {
168         read_input_line();
169         run_algorithm();
170         run_regression();
171         write_output_line();
172     }
173 }

```

2.8.4. src/dft.h

```

1 #ifndef _DFT_H_INCLUDED_
2 #define _DFT_H_INCLUDED_
3
4 #include "fourier.h"
5
6 class dft : public ft {
7     // private members
8
9     // protected members
10 protected:
11     virtual bool inverse();
12
13     virtual void run_algorithm();
14
15     // public members
16 public:
17     dft();
18
19     dft(istream *is);
20
21     dft(ostream *os);
22
23     dft(istream *is, ostream *os);
24
25     ~dft(){};
26 };
27

```

28 **#endif**

2.8.5. src/dft.cpp

```

1  #include <iomanip>
2  #include <iostream>
3
4  #include "../src/utls.h"
5  #include "../dft.h"
6
7  using namespace std;
8
9  dft::dft(): ft::ft() { }
10
11 dft::dft(istream *is): ft::ft(is) { }
12
13 dft::dft(ostream *os): ft::ft(os) { }
14
15 dft::dft(istream *is, ostream *os): ft::ft(is, os) { }
16
17 bool
18 dft::inverse() {
19     return false;
20 }
21
22 void
23 dft::run_algorithm() {
24     // NOTE: retorno rapido si no hay nada que procesar
25     // en el arreglo de input_.
26     if (input_.tamano() == 0) { return; }
27
28     // llevo tamano de entrada a una potencia de 2
29     // agregando 0s al final del arreglo
30     right_pad_input(input_);
31
32     double k, n, N = input_.tamano();
33     double arg, norm = get_norm();
34     complejo acum, j = get_exp_complejo();
35     lista<complejo>::iterador x;
36
37     for (k = 0 ; k < N; ++k) {
38         // arrancho en el primer elemento
39         x = input_.primero();
40         acum = 0;
41         n = 0;
42
43         // repito hasta el ultimo elemento de entrada
44         // la sumatoria de los  $x[n] * W(kn, N)$ 
45         do {
46             arg = 2 * M_PI * k * n / N;
47             acum += (x.dato()) * (cos(arg) + j.conjugado() * sin(arg));
48             n += 1;
49             x.avanzar();
50         } while ( !x.extremo() );
51
52         // multiplicamos por el normalizador que
53         // corresponda segun el modo
54         acum *= norm;
55
56         // escribo el acumulado a la salida una vez
57         // terminado de procesar el k-esimo elemento
58         // de la DFT/IDFT

```

```
59     output_.insertar_despues(acum, output_.ultimo());
60 }
61 }
```

2.8.6. src/idft.h

```
1  #ifndef _IDFT_H_INCLUDED_
2  #define _IDFT_H_INCLUDED_
3
4  #include "../dft.h"
5
6  class idft : public dft {
7      // private members
8
9      // protected members
10     protected:
11         virtual bool inverse();
12
13         virtual void run_algorithm();
14
15     // public members
16     public:
17         idft();
18
19         idft(istream *is);
20
21         idft(ostream *os);
22
23         idft(istream *is, ostream *os) ;
24
25         ~idft(){};
26 };
27
28 #endif
```

2.8.7. src/idft.cpp

```
1  #include <iomanip>
2  #include <iostream>
3
4  #include "../idft.h"
5
6  using namespace std;
7
8  idft::idft(): dft::dft() { }
9
10 idft::idft(istream *is): dft::dft(is) { }
11
12 idft::idft(ostream *os): dft::dft(os) { }
13
14 idft::idft(istream *is, ostream *os): dft::dft(is, os) { }
15
16 bool
17 idft::inverse() {
18     return true;
19 }
20
21 void
22 idft::run_algorithm() {
23     dft::run_algorithm();
24 }
```

2.8.8. src/fft.h

```

1  #ifndef _FFT_H_INCLUDED_
2  #define _FFT_H_INCLUDED_
3
4  #include "../fourier.h"
5
6  class fft : public ft {
7      // private members
8
9      // protected members
10     protected:
11         virtual bool inverse();
12
13         virtual void run_algorithm();
14
15         lista<complejo> recursive_algorithm(lista<complejo> &v);
16
17         void particion(lista<complejo> &v, lista<complejo> &even, lista<complejo> &odd);
18
19         lista<complejo> recompone(lista<complejo> &G, lista<complejo> &H, double const &N);
20
21     // public members
22     public:
23         fft();
24
25         fft(istream *is);
26
27         fft(ostream *os);
28
29         fft(istream *is, ostream *os);
30
31         ~fft(){};
32 };
33
34 #endif

```

2.8.9. src/fft.cpp

```

1  #include <iomanip>
2  #include <iostream>
3
4  #include "../src/utlis.h"
5  #include "../fft.h"
6
7  using namespace std;
8
9  fft::fft(): ft::ft() { }
10
11  fft::fft(istream *is): ft::ft(is) { }
12
13  fft::fft(ostream *os): ft::ft(os) { }
14
15  fft::fft(istream *is, ostream *os): ft::ft(is, os) { }
16
17  bool
18  fft::inverse() {
19      return false;
20  }
21
22  void
23  fft::run_algorithm() {

```

```

24 // NOTE: retorno rapido si no hay nada que procesar
25 // en el arreglo de input_.
26 if (input_.tamano() == 0) { return; }
27
28 // llevo tamano de entrada a una potencia de 2
29 // agregando 0s al final del arreglo
30 right_pad_input(input_);
31
32 // corro el algoritmo recursivo implementado
33 // desde el vector entrada al vector salidas
34 output_ = recursive_algorithm(input_);
35 }
36
37 lista<complejo>
38 fft::recursive_algorithm(lista<complejo> &v) {
39     int N = v.tamano();
40
41     if (N <= 1) {
42         return v;
43     }
44
45     lista<complejo> v_even_parts;
46     lista<complejo> v_odd_parts;
47     particion(v, v_even_parts, v_odd_parts);
48
49     lista<complejo> G = recursive_algorithm(v_even_parts);
50     lista<complejo> H = recursive_algorithm(v_odd_parts);
51
52     return recompone(G, H, N);
53 }
54
55 void
56 fft::particion(lista<complejo> &v, lista<complejo> &even, lista<complejo> &odd) {
57     std::size_t i = 0;
58     lista<complejo>::iterador it = v.primer();
59
60     do {
61         if (i % 2) {
62             odd.insertar_despues(it.dato(), odd.ultimo());
63         } else {
64             even.insertar_despues(it.dato(), even.ultimo());
65         }
66
67         i++;
68         it.avanzar();
69     } while (!it.extremo());
70 }
71
72 lista<complejo>
73 fft::recompone(lista<complejo> &G, lista<complejo> &H, double const &N) {
74     lista<complejo> X;
75
76     lista<complejo>::iterador it_G = G.primer();
77     lista<complejo>::iterador it_H = H.primer();
78
79     double arg;
80     complejo j = get_exp_complejo();
81     double norm = N == input_.tamano() ? get_norm() : 1; // solo en la ultima iteracion
82     complejo w;
83
84     // combine
85     // Para  $X[k]$  con  $0 < k < N/2$ 
86     for (int k = 0; k < N/2; ++k) {

```

```

87     arg = 2 * M_PI * k / N;
88     w = (cos(arg) + j.conjugado() * sin(arg));
89
90     complejo t = w * it_H.dato();
91     X.insertar_despues((it_G.dato() + t) * norm, X.ultimo());
92     if(!lit_G.extremo()) it_G.avanzar();
93     if(!lit_H.extremo()) it_H.avanzar();
94 }
95
96 it_G = G.primer();
97 it_H = H.primer();
98
99 //Para X[k] con N/2 < k < N
100 for (int k = 0; k < N/2; ++k) {
101     arg = 2 * M_PI * k / N;
102     w = (cos(arg) + j.conjugado() * sin(arg));
103
104     complejo t = w * it_H.dato();
105     X.insertar_despues((it_G.dato() - t) * norm, X.ultimo());
106     if(!lit_G.extremo()) it_G.avanzar();
107     if(!lit_H.extremo()) it_H.avanzar();
108 }
109
110 return X;
111 }

```

2.8.10. src/fft.h

```

1  #ifndef _IFFT_H_INCLUDED_
2  #define _IFFT_H_INCLUDED_
3
4  #include <fstream>
5  #include <sstream>
6
7  #include "./fft.h"
8
9  using namespace std;
10
11 class ifft : public fft {
12     // private members
13
14     // protected members
15     protected:
16         virtual bool inverse();
17
18         virtual void run_algorithm();
19
20     // public members
21     public:
22         ifft();
23
24         ifft(istream *is);
25
26         ifft(ostream *os);
27
28         ifft(istream *is, ostream *os);
29
30         ~ifft(){};
31 };
32
33 #endif

```


2.8.11. src/fft.cpp

```

1  #include <iomanip>
2  #include <iostream>
3  #include <fstream>
4  #include <sstream>
5
6  #include "../fft.h"
7
8  using namespace std;
9
10 ifft::ifft(): fft::fft() { }
11
12 ifft::ifft(istream *is): fft::fft(is) { }
13
14 ifft::ifft(ostream *os): fft::fft(os) { }
15
16 ifft::ifft(istream *is, ostream *os): fft::fft(is, os) { }
17
18 bool
19 ifft::inverse() {
20     return true;
21 }
22
23 void
24 ifft::run_algorithm() {
25     fft::run_algorithm();
26 }

```

2.8.12. src/utils.h

```

1  #ifndef _UTILS_H_INCLUDED_
2  #define _UTILS_H_INCLUDED_
3
4  #include <string>
5  #include <iomanip>
6  #include <sstream>
7  #include <iostream>
8
9  #include "../vendor/complejo.h"
10 #include "../vendor/lista.h"
11
12 using namespace std;
13
14 /**
15  * @brief Retorna un numero mayor al entero provisto
16  * que sea potencia de 2. La complejidad temporal
17  * de las operaciones de bitwise ordenadas es de
18  *  $O(\log [n])$ .
19  *
20  */
21 unsigned int next_power2 (unsigned int const &);
22 /**
23  * @brief Toma una lista de <complejo> y completa con
24  * ceros a la derecha segun la cantidad provista
25  * como segundo parametro.
26  *
27  */
28 void zero_pad(lista<complejo> &, unsigned int const &);
29 /**
30  * @brief Combina tanto next_power2 como zero_pad para
31  * dada una entrada de tipo lista<complejo>, com-

```

```

32 * pletar la misma con ceros hasta la proxima po-
33 * tencia de 2 la
34 *
35 */
36 void right_pad_input(lista<complejo> &);
37 /**
38 * @brief Dado un input stream y una lista<complejo>,
39 * lee hasta encontrar newline y almacena todo en
40 * la lista mencionada
41 *
42 */
43 void read_input_stream_line(istream *, lista<complejo> &);
44 /**
45 * @brief Dado un output stream y una lista<complejo>,
46 * escribe todos los elementos de la lista al stream
47 * concatenando un newline al terminar.
48 *
49 */
50 void write_output_stream_line(ostream *, lista<complejo> &);
51
52 #endif

```

2.8.13. src/utils.cpp

```

1
2 #include "../utils.h"
3
4 using namespace std;
5
6 unsigned int
7 next_power2 (unsigned int const &l) {
8     /*
9     * Si l es potencia de 2, (l-1) tendra solo 1s en los bits menos
10    * significativos, entonces "l & (l-1)" siempre sera 0.
11    * Se chequea aparte el caso l=0 dado que la expresion previa no aplica en
12    * dicho caso
13    */
14    bool isPowerOf2 = !(l == 0) && !(l & (l - 1));
15    if (isPowerOf2) {
16        return l;
17    }
18
19    unsigned int p = l;
20
21    /*
22    * Ya sabemos que l no es potencia de 2, entonces la proxima potencia de 2 sera el numero que
23    * tenga en '1' el bit "a la izquierda" del bit en '1' mas significativo del numero original, y
24    * los menos significativos en '0'. Por ej:
25    * 0010 0110 -> Numero original NO potencia de 2
26    * 0100 0000 -> Proxima potencia de 2 de dicho numero
27    *
28    * Para obtener dicho resultado, se completa con 1s a partir del bit mas significativo hacia
29    * el menos significativo, y luego se suma 1; con lo cual se propagara un bit de carry hasta
30    * setear en 1 el proximo bit del 1 que era mas significativo y dejando el resto en 0s.
31    *
32    * Por ej:
33    * 0010 0110 -> Numero original NO potencia de 2
34    * 0011 1111 -> Numero completado con 1s
35    * 0100 0000 -> Numero previo al que se le suma 1 obteniendo resultado final
36    *
37    * Para realizar el proceso de relleno con 1s, solo hace falta realizar ceil(log2(n))= operaciones de
38    * shift y OR, siendo "n" la cantidad de bits del numero.

```

```

39     * En nuestro caso consideramos  $n=32$ , entonces tenemos  $\text{ceil}(\log_2(32))=5$ 
40     */
41     p |= p >> 1;
42     p |= p >> 2;
43     p |= p >> 4;
44     p |= p >> 8;
45     p |= p >> 16;
46     p++;
47
48     return p;
49 }
50
51 void
52 right_pad_input(lista<complejo> &vector) {
53     // Si la cantidad de elementos del vector no es potencia de 2,
54     // agregamos 0s hasta completar tamaño con próxima potencia de 2
55     unsigned int tam = vector.tamano();
56     unsigned int v = next_power2(tam);
57     unsigned int zeros = v - tam;
58     zero_pad(vector, zeros);
59 }
60
61
62 void zero_pad(lista<complejo> &vector, unsigned int const &zeros) {
63     // NOTE: retorno si no debo agregar nada
64     if (zeros == 0) {
65         return;
66     }
67
68     for(int i = 0; i < zeros; i++){
69         complejo aux (0.0, 0.0);
70         vector.insertar_despues(aux, vector.ultimo());
71     }
72 }
73
74 void read_input_stream_line(istream *is, lista<complejo> &vector) {
75     complejo aux;
76     string line;
77
78     getline(*is, line);
79
80     stringstream linestream(line);
81
82     // leemos cada valor
83     while(linestream >> aux) {
84         vector.insertar_despues(aux, vector.ultimo());
85     }
86
87     // Error de formato en input stream.
88     // Detenemos la ejecucion del programa.
89     if (linestream.bad()) {
90         cerr << "cannot read from input stream." << endl;
91         exit(1);
92     }
93 }
94
95
96 void write_output_stream_line(ostream *os, lista<complejo> &vector) {
97     lista<complejo>::iterador out = vector.primer();
98
99     while(!out.extremo()) {
100         *os << out.dato() << " ";
101         out.avanzar();

```

```

102 }
103
104 *os << endl;
105 }

```

2.8.14. src/program.h

```

1  #ifndef _PROGRAM_H_INCLUDED_
2  #define _PROGRAM_H_INCLUDED_
3
4  class program {
5      // public members go here
6      public:
7          // constructor y destructor
8          program();
9          virtual ~program();
10
11         // devuelve el valor de codigo
12         // de la ejecucion del programa
13         int code();
14
15         // ejecuta el programa
16         virtual void compute() = 0;
17
18         // protected members go here
19         protected:
20             // almacena el valor del codigo
21             // de la ejecucion
22             int code_;
23
24             // settea el valor del codigo de la ejecucion
25             void code(const int &);
26
27         // private members go here
28     };
29
30 #endif

```

2.8.15. src/program.cpp

```

1  #include "program.h"
2
3  program::program(): code_(0) {}
4  program::~~program() {}
5
6  int program::code() {
7      return this->code_;
8  }
9
10 void program::code(const int &code) {
11     code_ = code;
12 }
13
14 void program::compute() { }

```

2.8.16. vendor/cmdline.h

```

1  #ifndef _CMDLINE_H_INCLUDED_
2  #define _CMDLINE_H_INCLUDED_

```

```

3
4 #include <string>
5 #include <iostream>
6
7 #define OPT_DEFAULT 0
8 #define OPT_SEEN 1
9 #define OPT_MANDATORY 2
10
11 struct option_t {
12     int has_arg;
13     const char *short_name;
14     const char *long_name;
15     const char *def_value;
16     void (*parse)(std::string const &); // Puntero a funcion de opciones
17     int flags;
18 };
19
20 class cmdline {
21     // Este atributo apunta a la tabla que describe todas
22     // las opciones a procesar. Por el momento, slo puede
23     // ser modificado mediante constructor, y debe finalizar
24     // con un elemento nulo.
25     //
26     option_t *option_table;
27
28     // El constructor por defecto cmdline::cmdline(), es
29     // privado, para evitar construir "parsers" (analizador
30     // sintctico, recibe una palabra y lo interpreta en
31     // una accin dependiendo su significado para el programa)
32     // sin opciones. Es decir, objetos de esta clase sin opciones.
33     //
34
35     cmdline();
36     int do_long_opt(const char *, const char *);
37     int do_short_opt(const char *, const char *);
38
39     public:
40         cmdline(option_t *);
41         void parse(int, char * const []);
42 };
43
44 #endif

```

2.8.17. vendor/cmdline.cpp

```

1 // cmdline – procesamiento de opciones en la linea de comando.
2 //
3 #include <string>
4 #include <cstdlib>
5 #include <iostream>
6 #include "cmdline.h"
7
8 using namespace std;
9
10 cmdline::cmdline() { }
11
12 /**
13  * Constructor con tabla
14  *
15  * Es lo mismo que hacer: 'option_table = table;'
16  *
17  * Siendo "option_table" un atributo de la clase cmdline

```

```

18  * y table un puntero a objeto o struct de "option_t".
19  *
20  * Se estaria contruyendo una instancia de la clase cmdline
21  * cargandole los datos que se hayan en table (la table con
22  * las opciones, ver el codigo en main.cpp)
23  */
24  cmdline::cmdline(option_t *table) : option_table(table) { }
25
26  void
27  cmdline::parse(int argc, char * const argv[]) {
28  #define END_OF_OPTIONS(p) \
29      ((p)->short_name == 0 \
30       && (p)->long_name == 0 \
31       && (p)->parse == 0)
32
33      // Primer pasada por la secuencia de opciones: marcamos
34      // todas las opciones, como no procesadas. Ver cdigo de
35      // abajo.
36      //
37      for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op)
38          op->flags &= ~OPT_SEEN;
39
40      // Recorremos el arreglo argv. En cada paso, vemos
41      // si se trata de una opcin corta, o larga. Luego,
42      // llamamos a la funcin de parseo correspondiente.
43      //
44      for (int i = 1; i < argc; ++i) {
45          // Todos los parmetros de este programa deben
46          // pasarse en forma de opciones. Encontrar un
47          // parametro no-opcin es un error.
48          //
49          if (argv[i][0] != '-') {
50              cerr << "Invalid non-option argument: "
51                  << argv[i]
52                  << endl;
53              exit(1);
54          }
55
56          // Usamos "--" para marcar el fin de las
57          // opciones; todo los argumentos que puedan
58          // estar a continuacin no son interpretados
59          // como opciones.
60          //
61          if (argv[i][1] == '-'
62              && argv[i][2] == 0)
63              break;
64
65          // Finalmente, vemos si se trata o no de una
66          // opcin larga; y llamamos al mtodo que se
67          // encarga de cada caso.
68          //
69          if (argv[i][1] == '-')
70              i += do_long_opt(&argv[i][2], argv[i + 1]);
71          else
72              i += do_short_opt(&argv[i][1], argv[i + 1]);
73      }
74
75      // Segunda pasada: procesamos aquellas opciones que,
76      // (1) no hayan figurado explcitamente en la lnea
77      // de comandos, y (2) tengan valor por defecto.
78      //
79      for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op) {
80  #define OPTION_NAME(op) \

```

```

81 (op->short_name ? op->short_name : op->long_name)
82 if (op->flags & OPT_SEEN)
83     continue;
84 if (op->flags & OPT_MANDATORY) {
85     cerr << "Option "
86         << "-"
87         << OPTION_NAME(op)
88         << " is mandatory."
89         << "\n";
90     exit(1);
91 }
92 if (op->def_value == 0)
93     continue;
94 op->parse(string(op->def_value));
95 }
96 }
97
98 int
99 cmdline::do_long_opt(const char *opt, const char *arg) {
100     // Recorremos la tabla de opciones, y buscamos la
101     // entrada larga que se corresponda con la opción de
102     // línea de comandos. De no encontrarse, indicamos el
103     // error.
104     //
105     for (option_t *op = option_table; op->long_name != 0; ++op) {
106         if (string(opt) == string(op->long_name)) {
107             // Marcamos esta opción como usada en
108             // forma explícita, para evitar tener
109             // que inicializarla con el valor por
110             // defecto.
111             //
112             op->flags |= OPT_SEEN;
113
114             if (op->has_arg) {
115                 // Como se trata de una opción
116                 // con argumento, verificamos que
117                 // el mismo haya sido provisto.
118                 //
119                 if (arg == 0) {
120                     cerr << "Option requires argument: "
121                         << "--"
122                         << opt
123                         << "\n";
124                     exit(1);
125                 }
126                 op->parse(string(arg));
127                 return 1;
128             } else {
129                 // Opción sin argumento.
130                 //
131                 op->parse(string(""));
132                 return 0;
133             }
134         }
135     }
136
137     // Error: opción no reconocida. Imprimimos un mensaje
138     // de error, y finalizamos la ejecución del programa.
139     //
140     cerr << "Unknown option: "
141         << "--"
142         << opt
143         << ". "

```

```

144     << endl;
145     exit(1);
146
147     // Algunos compiladores se quejan con funciones que
148     // lgicamente no pueden terminar, y que no devuelven
149     // un valor en esta ltima parte.
150     //
151     return -1;
152 }
153
154 int
155 cmdline::do_short_opt(const char *opt, const char *arg) {
156     option_t *op;
157
158     // Recorremos la tabla de opciones, y buscamos la
159     // entrada corta que se corresponda con la opcin de
160     // lnea de comandos. De no encontrarse, indicamos el
161     // error.
162     //
163     for (op = option_table; op->short_name != 0; ++op) {
164         if (string(opt) == string(op->short_name)) {
165             // Marcamos esta opcin como usada en
166             // forma explcita, para evitar tener
167             // que inicializarla con el valor por
168             // defecto.
169             //
170             op->flags |= OPT_SEEN;
171
172             if (op->has_arg) {
173                 // Como se trata de una opcin
174                 // con argumento, verificamos que
175                 // el mismo haya sido provisto.
176                 //
177                 if (arg == 0) {
178                     cerr << "Option requires argument: "
179                     << " _"
180                     << opt
181                     << "\n";
182                     exit(1);
183                 }
184                 op->parse(string(arg));
185                 return 1;
186             } else {
187                 // Opcin sin argumento.
188                 //
189                 op->parse(string(""));
190                 return 0;
191             }
192         }
193     }
194
195     // Error: opcin no reconocida. Imprimimos un mensaje
196     // de error, y finalizamos la ejecucin del programa.
197     //
198     cerr << "Unknown option: "
199     << " _"
200     << opt
201     << ". "
202     << endl;
203     exit(1);
204
205     // Algunos compiladores se quejan con funciones que
206     // lgicamente no pueden terminar, y que no devuelven

```



```

207 // un valor en esta ltima parte.
208 //
209 return -1;
210 }

```

2.8.18. vendor/complejo.h

```

1  #ifndef _COMPLEJO_H_INCLUDED_
2  #define _COMPLEJO_H_INCLUDED_
3
4  #include <iostream>
5  #include <iomanip>
6
7  class complejo {
8      double re_, im_;
9
10     public:
11         complejo();
12         complejo(double);
13         complejo(double, double);
14         complejo(const complejo &);
15         complejo const &operator=(complejo const &);
16         complejo const &operator*=(complejo const &);
17         complejo const &operator+=(complejo const &);
18         complejo const &operator-=(complejo const &);
19         ~complejo();
20
21         double re() const;
22         double im() const;
23         double abs() const;
24         double abs2() const;
25         complejo const &conjugar();
26         complejo const conjugado() const;
27         bool zero() const;
28
29         friend complejo const operator+(complejo const &, complejo const &);
30         friend complejo const operator-(complejo const &, complejo const &);
31         friend complejo const operator*(complejo const &, complejo const &);
32         friend complejo const operator/(complejo const &, complejo const &);
33         friend complejo const operator/(complejo const &, double);
34
35         friend bool operator==(complejo const &, double);
36         friend bool operator==(complejo const &, complejo const &);
37
38         friend std::ostream &operator<<(std::ostream &, const complejo &);
39         friend std::istream &operator>>(std::istream &, complejo &);
40 };
41
42 #endif

```

2.8.19. vendor/complejo.cpp

```

1  #include <iostream>
2  #include <cmath>
3  #include "complejo.h"
4
5  using namespace std;
6
7  complejo::complejo() : re_(0), im_(0) { }
8
9  complejo::complejo(double r) : re_(r), im_(0) { }

```

```
10
11 complejo::complejo(double r, double i) : re_(r), im_(i) { }
12
13 complejo::complejo(complejo const &c) : re_(c.re_), im_(c.im_) { }
14
15 complejo const &
16 complejo::operator=(complejo const &c) {
17     re_ = c.re_;
18     im_ = c.im_;
19     return *this;
20 }
21
22 complejo const &
23 complejo::operator*=(complejo const &c) {
24     double re = re_ * c.re_
25             - im_ * c.im_;
26     double im = re_ * c.im_
27             + im_ * c.re_;
28     re_ = re, im_ = im;
29     return *this;
30 }
31
32 complejo const &
33 complejo::operator+=(complejo const &c) {
34     double re = re_ + c.re_;
35     double im = im_ + c.im_;
36     re_ = re, im_ = im;
37     return *this;
38 }
39
40 complejo const &
41 complejo::operator-=(complejo const &c) {
42     double re = re_ - c.re_;
43     double im = im_ - c.im_;
44     re_ = re, im_ = im;
45     return *this;
46 }
47
48 complejo::~complejo() { }
49
50 double
51 complejo::re() const {
52     return re_;
53 }
54
55 double complejo::im() const {
56     return im_;
57 }
58
59 double
60 complejo::abs() const {
61     return std::sqrt(re_ * re_ + im_ * im_);
62 }
63
64 double
65 complejo::abs2() const {
66     return re_ * re_ + im_ * im_;
67 }
68
69 complejo const &
70 complejo::conjugar() {
71     im_ *= -1;
72     return *this;
```

```

73 }
74
75 complejo const
76 complejo::conjugado() const {
77     return complejo(re_, -im_);
78 }
79
80 bool
81 complejo::zero() const {
82     #define ZERO(x) ((x) == +0.0 && (x) == -0.0)
83     return ZERO(re_) && ZERO(im_) ? true : false;
84 }
85
86 complejo const
87 operator+(complejo const &x, complejo const &y) {
88     complejo z(x.re_ + y.re_, x.im_ + y.im_);
89     return z;
90 }
91
92 complejo const
93 operator-(complejo const &x, complejo const &y) {
94     complejo r(x.re_ - y.re_, x.im_ - y.im_);
95     return r;
96 }
97
98 complejo const
99 operator*(complejo const &x, complejo const &y) {
100     complejo r(x.re_ * y.re_ - x.im_ * y.im_,
101               x.re_ * y.im_ + x.im_ * y.re_);
102     return r;
103 }
104
105 complejo const
106 operator/(complejo const &x, complejo const &y) {
107     return x * y.conjugado() / y.abs2();
108 }
109
110 complejo const
111 operator/(complejo const &c, double f) {
112     return complejo(c.re_ / f, c.im_ / f);
113 }
114
115 bool
116 operator==(complejo const &c, double f) {
117     bool b = (c.im_ != 0 || c.re_ != f) ? false : true;
118     return b;
119 }
120
121 bool
122 operator==(complejo const &x, complejo const &y) {
123     bool b = (x.re_ != y.re_ || x.im_ != y.im_) ? false : true;
124     return b;
125 }
126
127 ostream &
128 operator<<(ostream &os, const complejo &c) {
129     // prints: {c.re_}, {c.im_}
130     return os << "(" << c.re_ << ", " << c.im_ << ")";
131 }
132
133 istream &
134 operator>>(istream &is, complejo &c) {
135     int good = false;

```

```

136  int bad = false;
137  double re = 0;
138  double im = 0;
139  char ch = 0;
140
141  if (is >> ch
142      && ch == '(') {
143      if (is >> re
144          && is >> ch
145          && ch == ','
146          && is >> im
147          && is >> ch
148          && ch == ')')
149          good = true;
150      else
151          bad = true;
152  } else if (is.good()) {
153      is.putback(ch);
154      if (is >> re)
155          good = true;
156      else
157          bad = true;
158  }
159
160  if (good)
161      c.re_ = re, c.im_ = im;
162  if (bad)
163      is.clear(ios::badbit);
164
165  return is;
166  }

```

2.8.20. vendor/lista.h

```

1  // lista.h – Implementación de listas doblemente enlazadas, no circulares,
2  // usando templates e iteradores.
3  //
4
5
6  #ifndef LISTA_H_
7  #define LISTA_H_
8
9  #include <cstdlib>
10
11 template<typename T>
12 class lista
13 {
14     // Ponemos la clase nodo dentro de la parte privada de la lista, con la
15     // idea de ocultar su existencia. La idea es reemplazar las funciones
16     // normalmente destinadas a los nodos por operaciones provistas por
17     // el TDA iterador. Ver ms abajo.
18     //
19     class nodo
20     {
21         // Debido al fuerte acople entre iteradores y la estructura a
22         // la cual iteran (es decir, el iterador necesita conocer los
23         // detalles de implementación del TDA para poder abstraer el
24         // recorrido), permitimos que la clase iterador tenga acceso
25         // directo a los detalles internos del TDA.
26         friend class iterador;
27         friend class lista;
28     }

```

```

29     nodo *sig_;
30     nodo *ant_;
31     T dato_;
32
33 public:
34     nodo(T const&);
35     ~nodo();
36 };
37
38 // Los atributos de cada instancia de la clase lista son enlaces a
39 // dos nodos distinguidos de la secuencia: primero y ltimo. Esto es
40 // necesario para poder tener acceso inmediato, en todo momento, a
41 // estos elementos, y reducir la complejidad de las operaciones que
42 // necesiten hacerlo (por ejemplo, lista::primero y lista::ultimo).
43 //
44 nodo *pri_;
45 nodo *ult_;
46 size_t tam_;
47
48 public:
49 // Iterador de lista. Buscamos proveer una metodologa eficiente para
50 // recorrer, uno a uno, los elementos de la estructura, evitando la
51 // manipulacin explcita de la misma. En este caso, adems, queremos
52 // usarlos como cursores, indicando los elementos a insertar o borrar
53 // de la lista.
54 //
55 class iterador
56 {
57     friend class lista;
58
59     nodo *actual_;
60     iterador(nodo*);
61
62 public:
63     iterador();
64     iterador(lista<T> const &);
65     iterador(iterador const &);
66     ~iterador();
67
68     T& dato();
69     T const &dato() const;
70     iterador &avanzar();
71     iterador &retroceder();
72     bool extremo() const;
73
74     bool operator==(const iterador &) const;
75     bool operator!=(const iterador &) const;
76     iterador const &operator=(iterador const &);
77 };
78
79 typedef T t_dato;
80 typedef nodo t_nodo;
81 typedef iterador t_iter;
82
83 lista();
84 lista(const lista &);
85 ~lista();
86
87 // Mtodos pblicos, especficos del TDA lista.
88 //
89 size_t tamano() const;
90 bool contiene(const T &) const;
91 bool vacia() const;

```

```

92 void insertar(const T &);
93 void insertar_antes(const T &, iterador const &);
94 void insertar_despues(const T &, iterador const &);
95 void borrar(const T &);
96 void clear();
97 lista const &operator=(lista const &);
98
99 // Esta clase provee mtodos para generar iteradores preposicionados
100 // en los nodos distinguidos (i.e. inicial y final) de la secuencia.
101 // Combinando esto con las operaciones de comparacin provistas por
102 // el TDA iterador, buscamos facilitar las expresiones de corte o
103 // continuacin de los ciclos de iteracin. Por ejemplo,
104 //
105 // lista<int>::iterador it;
106 // ...
107 // for( it = L.ultimo(); !it.extremo(); it.retroceder()) {
108 // // ...
109 // }
110 //
111 // Tambin,
112 //
113 // lista<int>::iterador it;
114 // lista<int>::iterador mitad;
115 // ...
116 // for(it = L.primer(); it != mitad; it.avanzar()) {
117 // // ...
118 // }
119 //
120 iterador primero() const;
121 iterador ultimo() const;
122
123 };
124
125 template<typename T>
126 lista<T>::iterador::iterador() : actual_(0)
127 {
128 }
129
130 template<typename T>
131 lista<T>::iterador::iterador(nodo *actual) : actual_(actual)
132 {
133 }
134
135 template<typename T>
136 lista<T>::iterador::iterador(lista<T> const &l) : actual_(l.pri_)
137 {
138 }
139
140 template<typename T>
141 lista<T>::iterador::iterador(iterador const &it) : actual_(it.actual_)
142 {
143 }
144
145 template<typename T>
146 lista<T>::iterador::~~iterador()
147 {
148 }
149
150 template<typename T>
151 T &lista<T>::iterador::dato()
152 {
153     return actual_>dato_;
154 }

```

```

155
156 template<typename T>
157 T const &lista<T>::iterador::dato() const
158 {
159     return actual_ -> dato_;
160 }
161
162 template<typename T>
163 typename lista<T>::iterador &lista<T>::iterador::avanzar()
164 {
165     if (actual_)
166         actual_ = actual_ -> sig_;
167     return *this;
168 }
169
170 template<typename T>
171 typename lista<T>::iterador &lista<T>::iterador::retroceder()
172 {
173     if (actual_)
174         actual_ = actual_ -> ant_;
175     return *this;
176 }
177
178 template<typename T>
179 bool lista<T>::iterador::extremo() const
180 {
181     return actual_ == 0 ? true : false;
182 }
183
184 template<typename T>
185 bool lista<T>::iterador::operator==(const typename lista<T>::iterador &it2) const
186 {
187     return actual_ == it2.actual_;
188 }
189
190 template<typename T>
191 bool lista<T>::iterador::operator!=(const typename lista<T>::iterador &it2) const
192 {
193     return actual_ != it2.actual_;
194 }
195
196 template<typename T>
197 typename lista<T>::iterador const &lista<T>::iterador::operator=(iterador const &orig)
198 {
199     if (this != &orig)
200         actual_ = orig.actual_;
201     return *this;
202 }
203
204 template<typename T>
205 lista<T>::nodo::nodo(const T &t) : sig_(0), ant_(0), dato_(t)
206 {
207 }
208
209 template<typename T>
210 lista<T>::nodo::~~nodo()
211 {
212 }
213
214 template<typename T>
215 lista<T>::lista() : pri_(0), ult_(0), tam_(0)
216 {
217 }

```

```

218
219 template<typename T>
220 lista<T>::lista(const lista &orig) : pri_(0), ult_(0), tam_(orig.tam_)
221 {
222     nodo *iter;
223     nodo *ant;
224
225     // Recorremos la secuencia original en sentido directo. En cada paso,
226     // creamos un nodo, copiando el dato correspondiente, y lo enganchamos
227     // al final de nuestra nueva lista.
228     //
229     for (iter = orig.pri_, ant = 0; iter != 0; iter = iter->sig_)
230     {
231         // Creamos un nodo, copiando el dato, y lo enganchamos en e
232         // final de nuestra lista.
233         //
234         nodo *nuevo = new nodo(iter->dato_);
235         nuevo->ant_ = ant;
236         nuevo->sig_ = 0;
237
238         // Si sta no es la primera pasada, es decir, si no se trata
239         // del primer nodo de la lista, ajustamos el enlace sig_ del
240         // nodo anterior.
241         //
242         if (ant != 0)
243             ant->sig_ = nuevo;
244
245         // Adems, tenemos que ajustar los punteros a los elementos
246         // distinguidos de la secuencia, primero y ltimo. En el caso
247         // de pri_ (enlace al primer elemento), esto lo vamos a
248         // hacer una nica vez; para el caso de ult_, iremos tomando
249         // registro del ltimo nodo procesado, para ajustarlo antes
250         // de retornar.
251         //
252         if (pri_ == 0)
253             pri_ = nuevo;
254         ant = nuevo;
255     }
256
257     // Ajustamos el puntero al ltimo elemento de la copia.
258     ult_ = ant;
259 }
260
261 template<typename T>
262 lista<T>::~~lista()
263 {
264     for (nodo *p = pri_; p; )
265     {
266         nodo *q = p->sig_;
267         delete p;
268         p = q;
269     }
270 }
271
272 template<typename T>
273 size_t lista<T>::tamano() const
274 {
275     return tam_;
276 }
277
278 template<typename T>
279 bool lista<T>::contiene(const T &elem) const
280 {

```



```

281     nodo *iter;
282
283     for (iter = pri_; iter != 0; iter = iter->sig_)
284         if (elem == iter->dato_)
285             return true;
286     return false;
287 }
288
289 template<typename T>
290 bool lista<T>::vacía() const
291 {
292     return pri_ ? false : true;
293 }
294
295 template<typename T>
296 void lista<T>::insertar(const T &t)
297 {
298     nodo *p = new nodo(t);
299     p->sig_ = pri_;
300     p->ant_ = 0;
301
302     if (pri_)
303         pri_->ant_ = p;
304     pri_ = p;
305     if (!ult_)
306         ult_ = p;
307
308     tam_++;
309 }
310
311 template<typename T>
312 void lista<T>::insertar_despues(const T &t, iterador const &it)
313 {
314     nodo *nuevo = new nodo(t);
315     nodo *actual = it.actual_;
316
317     if (actual == 0)
318     {
319         if (pri_ != 0)
320             std::abort();
321         pri_ = nuevo;
322         ult_ = nuevo;
323     }
324     else
325     {
326         if (actual->sig_ != 0)
327             actual->sig_->ant_ = nuevo;
328         nuevo->sig_ = actual->sig_;
329         nuevo->ant_ = actual;
330         actual->sig_ = nuevo;
331         if (ult_ == actual)
332             ult_ = nuevo;
333     }
334
335     tam_++;
336 }
337
338 template<typename T>
339 void lista<T>::insertar_antes(const T &t, iterador const &it)
340 {
341     nodo *nuevo = new nodo(t);
342     nodo *actual = it.actual_;
343

```

```

344     if (actual == 0)
345     {
346         if (pri_ != 0)
347             std::abort();
348         pri_ = nuevo;
349         ult_ = nuevo;
350     }
351     else
352     {
353         if (actual->ant_ != 0)
354             actual->ant_->sig_ = nuevo;
355         nuevo->sig_ = actual;
356         nuevo->ant_ = actual->ant_;
357         actual->ant_ = nuevo;
358         if (pri_ == actual)
359             pri_ = nuevo;
360     }
361
362     tam_++;
363 }
364
365 template<typename T>
366 void lista<T>::borrar(const T &t)
367 {
368     nodo *iter, *sig=0;
369
370     for (iter = pri_; iter != 0; iter = sig)
371     {
372         sig = iter->sig_;
373         if (t == iter->dato_)
374         {
375             nodo *ant = iter->ant_;
376             if (ant == 0)
377                 pri_ = sig;
378             else
379                 ant->sig_ = sig;
380             if (sig == 0)
381                 ult_ = ant;
382             else
383                 sig->ant_ = ant;
384             delete iter;
385
386             tam_--;
387         }
388     }
389 }
390
391 template<typename T>
392 void lista<T>::clear()
393 {
394     nodo *iter, *sig=0;
395
396     for (iter = pri_; iter != 0; iter = sig)
397     {
398         sig = iter->sig_;
399
400         nodo *ant = iter->ant_;
401         if (ant == 0)
402             pri_ = sig;
403         else
404             ant->sig_ = sig;
405         if (sig == 0)
406             ult_ = ant;

```

```

407     else
408         sig->ant_ = ant;
409     delete iter;
410
411     tam_--;
412
413 }
414 }
415
416 template<typename T>
417 typename lista<T>::iterador lista<T>::ultimo() const
418 {
419     return typename lista<T>::iterador(ult_);
420 }
421
422 template<typename T>
423 typename lista<T>::iterador lista<T>::primero() const
424 {
425     return typename lista<T>::iterador(pri_);
426 }
427
428 template<typename T>
429 lista<T> const &lista<T>::operator=(lista const &orig)
430 {
431     nodo *iter;
432     nodo *sig;
433     nodo *ant;
434
435     if (this != &orig)
436     {
437         for (iter = pri_; iter != 0; )
438         {
439             sig = iter->sig_;
440             delete iter;
441             iter = sig;
442         }
443
444         pri_ = 0;
445         ult_ = 0;
446
447         for (iter = orig.pri_, ant = 0; iter != 0; iter = iter->sig_)
448         {
449             nodo *nuevo = new nodo(iter->dato_);
450             nuevo->ant_ = ant;
451             nuevo->sig_ = 0;
452             if (ant != 0)
453                 ant->sig_ = nuevo;
454             if (pri_ == 0)
455                 pri_ = nuevo;
456             ant = nuevo;
457         }
458         ult_ = ant;
459         tam_ = orig.tam_;
460     }
461
462     return *this;
463 }
464
465
466 #endif

```

2.8.21. test/tests.sh

```
1  #!/bin/bash
2  parent_path=$( cd "$(dirname "${BASH_SOURCE[0]}")" ; pwd -P )
3  bin="$parent_path/./bin/tp2"
4
5  garbagefile="$parent_path/files/_garbage.txt"
6  testfile="$parent_path/files/test_1_signals_10000_points.txt"
7  testfile2="$parent_path/files/test_10_signals_1000_points.txt"
8  testfile3="$parent_path/files/test_100_signals_100_points.txt"
9
10 test_fft_1_signal_10000_points() {
11     echo "FFT 1 signal 10000 points x100 runs"
12
13     max=100
14     for i in `seq 2 $max`
15     do
16         $bin -i $testfile -o $garbagefile
17     done
18 }
19
20 test_fft_10_signal_1000_points() {
21     echo "FFT 10 signals 1000 points x100 runs"
22
23     max=100
24     for i in `seq 2 $max`
25     do
26         $bin -i $testfile2 -o $garbagefile
27     done
28 }
29
30 test_fft_100_signal_100_points() {
31     echo "FFT 100 signals 100 points x100 runs"
32
33     max=100
34     for i in `seq 2 $max`
35     do
36         $bin -i $testfile3 -o $garbagefile
37     done
38 }
39
40 test_dft_1_signal_10000_points() {
41     echo "DFT 1 signal 10000 points x10 runs"
42
43     max=10
44     for i in `seq 2 $max`
45     do
46         $bin -m DFT -i $testfile -o $garbagefile
47     done
48 }
49
50 test_dft_10_signal_1000_points() {
51     echo "DFT 10 signals 1000 points x10 runs"
52
53     max=10
54     for i in `seq 2 $max`
55     do
56         $bin -m DFT -i $testfile2 -o $garbagefile
57     done
58 }
59
60 test_dft_100_signal_100_points() {
61     echo "DFT 100 signals 100 points x10 runs"
62
63     max=10
```

```

64   for i in `seq 2 $max`
65   do
66       $bin -m DFT -i $testfile3 -o $garbagefile
67   done
68 }
69
70 echo ""
71 time test_fft_1_signal_10000_points
72 echo "Done."
73
74 echo ""
75 time test_fft_10_signal_1000_points
76 echo "Done."
77
78 echo ""
79 time test_fft_100_signal_100_points
80 echo "Done."
81
82 echo ""
83 time test_dft_1_signal_10000_points
84 echo "Done."
85
86 echo ""
87 time test_dft_10_signal_1000_points
88 echo "Done."
89
90 echo ""
91 time test_dft_100_signal_100_points
92 echo "Done."

```

2.8.22. test/files/test₁signals₁0000_points.txt

[illegible]

2.8.23. `test/files/test10signals1000points.txt`

[illegible]

[illegible]

[illegible]

[illegible]

2.8.24. `test/files/test100signals100oints.txt`

[illegible]

[illegible]

Pág 58

Pág 59

[illegible]

2.8.25. test/results/perf-total.txt

```

1 FFT 1 signal 10000 points x100 runs
2
3 real 0m15.088s
4 user 0m13.685s
5 sys 0m0.491s
6 Done.
7
8 FFT 10 signals 1000 points x100 runs
9
10 real 0m7.329s
11 user 0m6.784s
12 sys 0m0.382s
13 Done.
14
15 FFT 100 signals 100 points x100 runs
16
17 real 0m7.216s
18 user 0m6.641s
19 sys 0m0.436s
20 Done.
21
22 DFT 1 signal 10000 points x10 runs
23
24 real 4m44.991s
25 user 4m38.289s
26 sys 0m1.748s
27 Done.
28
29 DFT 10 signals 1000 points x10 runs
30
31 real 0m11.419s
32 user 0m11.125s
33 sys 0m0.113s
34 Done.
35
36 DFT 100 signals 100 points x10 runs
37
38 real 0m2.051s
39 user 0m1.925s
40 sys 0m0.069s
41 Done.

```

2.8.26. test/results/perf-lectura.txt

```

1 FFT 1 signal 10000 points x100 runs
2
3 real 0m1.750s
4 user 0m0.616s
5 sys 0m0.269s
6 Done.
7
8 FFT 10 signals 1000 points x100 runs

```

```
9
10 real 0m1.015s
11 user 0m0.614s
12 sys 0m0.259s
13 Done.
14
15 FFT 100 signals 100 points x100 runs
16
17 real 0m1.070s
18 user 0m0.644s
19 sys 0m0.281s
20 Done.
21
22 DFT 1 signal 10000 points x10 runs
23
24 real 0m0.120s
25 user 0m0.064s
26 sys 0m0.036s
27 Done.
28
29 DFT 10 signals 1000 points x10 runs
30
31 real 0m0.103s
32 user 0m0.059s
33 sys 0m0.031s
34 Done.
35
36 DFT 100 signals 100 points x10 runs
37
38 real 0m0.106s
39 user 0m0.060s
40 sys 0m0.034s
41 Done.
```

2.8.27. test/results/perf-lectura-y-alg.txt

```
1 FFT 1 signal 10000 points x100 runs
2
3 real 0m13.770s
4 user 0m12.356s
5 sys 0m0.421s
6 Done.
7
8 FFT 10 signals 1000 points x100 runs
9
10 real 0m6.445s
11 user 0m5.982s
12 sys 0m0.321s
13 Done.
14
15 FFT 100 signals 100 points x100 runs
16
17 real 0m5.984s
18 user 0m5.568s
19 sys 0m0.276s
20 Done.
21
22 DFT 1 signal 10000 points x10 runs
23
24 real 4m35.349s
25 user 4m30.603s
26 sys 0m0.849s
```

```

27 Done.
28
29 DFT 10 signals 1000 points x10 runs
30
31 real 0m10.515s
32 user 0m10.413s
33 sys 0m0.049s
34 Done.
35
36 DFT 100 signals 100 points x10 runs
37
38 real 0m1.784s
39 user 0m1.725s
40 sys 0m0.034s
41 Done.

```

2.8.28. Makefile

```

1 PROJ = tp2
2 SRCS = $(shell find src vendor -name "*.cpp" | xargs)
3 ENTRY = main.cpp
4 OUTDIR = bin
5 OUTFILE = $(OUTDIR)/$(PROJ)
6 CXXARGS = -Wall -g
7 CXXFLAGS = -Isrc -Ivendor -I. -isystem $(CXXARGS)
8 LDFLAGS =
9 CXX = g++
10
11 build: bin/tp2
12
13 bin/tp2:
14     $(CXX) $(CXXFLAGS) $(ENTRY) $(SRCS) -o $(OUTFILE)
15
16 test: build
17     ./test/tests.sh
18
19 test-valgrind: bin/tp2
20     # memcheck is default --tool
21     valgrind --leak-check=yes $(OUTFILE) -i ./test/files/test_1_signals_10000_points.txt -o ./test/files/_garbage.
22         ↪ txt
23     valgrind --leak-check=yes $(OUTFILE) -i ./test/files/test_10_signals_1000_points.txt -o ./test/files/_garbage.
24         ↪ txt
25     valgrind --leak-check=yes $(OUTFILE) -i ./test/files/test_100_signals_100_points.txt -o ./test/files/_garbage.
26         ↪ txt
27
28 .PHONY: bin/tp2

```

2.8.29. Makefile.WIN32

```

1 PROJ = tp2
2 SRCS = $(wildcard ./src/*.cpp ./vendor/*.cpp)
3 ENTRY = main.cpp
4 OUTDIR = bin
5 OUTFILE = $(OUTDIR)/$(PROJ)
6 CXXARGS = -static-libgcc -static-libstdc++ -Wall -g
7 CXXFLAGS = -Isrc -Ivendor -I. -isystem $(CXXARGS)
8 LDFLAGS =
9 CXX = g++
10
11 build: bin/tp2
12

```

```
13 bin/tp2:
14   $(CXX) $(CXXFLAGS) $(ENTRY) $(SRCS) -o $(OUTFILE)
15 test: build
16   ./test/tests.sh
17
18 .PHONY: bin/tp2
```