

Algoritmos y Programación II
TP1: Recursividad

Bourbon, Rodrigo
Carreño Romano, Carlos Germán
Sampayo, Sebastián Lucas

Primer Cuatrimestre de 2015



**FACULTAD
DE INGENIERIA**

Universidad de Buenos Aires

Índice

1. Objetivos	1
2. Introducción	1
2.1. Transformada de Fourier	1
2.1.1. Transformada discreta de Fourier	1
2.1.2. Transformada rápida de Fourier	1
2.2. Relleno con ceros (<i>zero padding</i>)	1
3. Standard de estilo	1
4. Diseño del programa	1
5. Opciones del programa	2
6. Métodos de la Transformada	2
6.1. FFT	2
6.1.1. Complejidad Temporal	2
6.2. DFT	4
6.2.1. Complejidad Temporal	4
7. Estructura de archivos	6
8. Compilación	7
9. Casos de prueba	7
9.1. Caso 1	8
9.2. Caso 2	9
9.3. Caso 3	9
9.4. Caso 4	9
10. Código	10
10.1. Programa principal	10
10.1.1. main.cc	10
10.2. Parseo de opciones del programa	11
10.2.1. types.h	11
10.2.2. cmdline.h	12
10.2.3. cmdline.cc	12
10.2.4. arguments.h	16
10.2.5. arguments.cc	17
10.3. Clase Vector	19
10.3.1. vector.h	19
10.3.2. vector.cc	20
10.4. Clase Complejo	23
10.4.1. complex.h	23
10.4.2. complex.cc	24
10.5. Funciones utilitarias	28
10.5.1. utilities.h	28
10.5.2. utilities.cc	28
10.6. Métodos para Transformar	29
10.6.1. dft_methods.h	29
10.6.2. dft_methods.cc	30
11. Enunciado	34

1. Objetivos

Ejercitar técnicas de diseño, análisis, e implementación de algoritmos recursivos.

2. Introducción

2.1. Transformada de Fourier

La transformada de Fourier es una operación matemática que transforma una señal de dominio de tiempo a dominio de frecuencia y viceversa. Tiene muchas aplicaciones en la ingeniería, especialmente para la caracterización frecuencial de señales y sistemas lineales. Es decir, la transformada de Fourier se utiliza para conocer las características frecuenciales de las señales y el comportamiento de los sistemas lineales ante estas señales.

2.1.1. Transformada discreta de Fourier

Una *DFT* (Transformada de Fourier Discreta - por sus siglas en inglés) es el nombre dado a la transformada de Fourier cuando se aplica a una señal digital (discreta) en vez de una analógica (continua).

2.1.2. Transformada rápida de Fourier

Una *FFT* (Transformada Rápida de Fourier - por sus siglas en inglés) es una versión más rápida de la *DFT* que puede ser aplicada cuando el número de muestras de la señal es una potencia de dos. Un cálculo de *FFT* toma aproximadamente $N \log(N)$ operaciones, mientras que *DFT* toma aproximadamente N^2 operaciones, así es que la *FFT* es significativamente más rápida.

2.2. Relleno con ceros (*zero padding*)

Esto significa que se agregarán ceros al final de la secuencia de valores ingresados. En el presente trabajo se decidió completar con ceros las muestras leídas en la entrada del programa hasta llevarlas a la potencia de dos más cercana. Esta adición no afecta el espectro de frecuencia de la señal y es recomendable ya que se acelera el cálculo de *FFT*. El relleno de ceros también incrementa la resolución de la frecuencia de una *FFT*.

3. Standard de estilo

Adoptamos la convención de estilo de código de Google para C++, salvando las siguientes excepciones:

- Streams: utilizamos flujos de entrada y salida
- Sobrecarga de operadores

<https://google-styleguide.googlecode.com/svn/trunk/cppguide.html#Naming>

4. Diseño del programa

Para resolver el problema, se optó por un diseño *top-down*, es decir, planteando el algoritmo de alto nivel con un diagrama en bloques. Luego se implementó cada bloque por separado para que cumpla con las necesidades de entrada y salida. Una vez hecho esto, los bloques se interconectan en el programa principal (*main*).

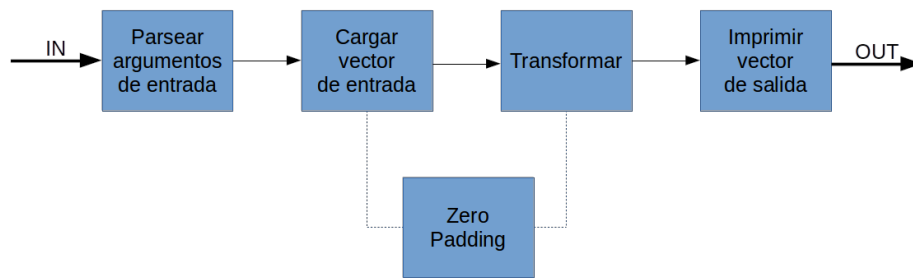


Figura 1: Diagrama en bloques del programa principal.

5. Opciones del programa

El programa se ejecuta en línea de comandos, y las opciones que admite (sin importar el orden de aparición) son las siguientes:

nombre largo (nombre corto): descripción

- **--input (-i):**

En esta opción se indica un argumento que debe ser la ruta de un archivo del cual queramos leer o bien la opción por defecto "-" que utiliza el flujo de entrada estándar.

- **--output (-o):**

En esta opción se indica un argumento que debe ser la ruta de un archivo en el cual queramos imprimir o bien la opción por defecto "-" que utiliza el flujo de salida estándar.

- **--method (-m):**

En esta opción se indica la acción que se debe realizar sobre los datos de la entrada, estos pueden ser:

- Transformada discreta de Fourier (**-dft**).
- Transformada discreta inversa de Fourier (**-idft**).
- Transformada rápida de Fourier (**-fft**).
- Transformada rápida inversa de Fourier (**-ifft**).

Por defecto el programa se ejecuta con la transformada rápida de fourier.

6. Métodos de la Transformada

como fue implementado dft y fft, funciones genéricas, máscaras, complejidad temporal, espacial, etc.

6.1. FFT

6.1.1. Complejidad Temporal

Para estudiar el costo temporal de esta implementación — $T(N)$ — se analizó cada línea de código de la función `calculate_fft_generic()`.

Al principio, todas las sentencias son de orden constante hasta que aparece el primer ciclo:

```

1 static Vector<Complex>
2 calculate_fft_generic(Vector<Complex> const &x, bool inverse)
3 {
4     size_t N;
5     N = x.size();
6 
```

```

7  Vector<Complex> X(N);
8
9  // Por defecto se calcula la FFT con estos parámetros:
10 double factor = 1;
11 int W_phase_sign = -1;
12
13 // En caso de tener que calcular la inversa,
14 // modifiko el factor de escala y el signo de la fase de W.
15 if (inverse)
16 {
17     factor = 1.0/N;
18     W_phase_sign = 1;
19 }
20
21 if (N > 1)
22 {
23     // Divido el problema en 2:
24     // Suponemos que la entrada es par y potencia de 2
25     Vector<Complex> p(N/2);
26     Vector<Complex> q(N/2);
27     Vector<Complex> P(N/2);
28     Vector<Complex> Q(N/2);

```

Las únicas expresiones que ofrecen cierta duda de que su coste sea constante son las últimas —constructores de $N/2$ elementos. Sin embargo, al ver la implementación de dicho constructor no quedan dudas, ya que solo consiste en una comparación, una asignación, y una llamada a *new*:

```

1  template <typename T>
2  Vector<T>::Vector(int s)
3  {
4      if (s <= 0)
5      {
6          exit(1);
7      }
8      else
9      {
10         size_ = s;
11         ptr_ = new T[size_];
12     }
13 }

```

Continuando con la función *calculate_fft_generic()* :

```

1      for (size_t i=0; i<N/2; i++)
2      {
3          p[i] = x[2*i];
4          q[i] = x[2*i+1];
5      }
6
7      P = calculate_fft(p);
8      Q = calculate_fft(q);
9
10     // Combino las soluciones:
11     for (size_t k=0; k<N; k++)
12     {
13         Complex W(cos(k*(2*PI)/N),
14                   W_phase_sign*sin(k*(2*PI)/N));
15         // Para que se repitan los elementos cíclicamente, se utiliza la función
           módulo

```

```

16         size_t k2 = k % (N/2);
17
18         X[k] = factor * (P[k2] + W*Q[k2]);
19     }
20 }
21 else
22 {
23     X = x;
24 }
25
26 return X;
27 }

```

Se tiene un ciclo de $N/2$ iteraciones cuyas operaciones en cada caso son de orden constante, con lo cual el orden de este ciclo es $\mathcal{O}(N/2)$.

A continuación encontramos las llamadas recursivas. Dado que el tamaño de la entrada se reduce a la mitad, tenemos 2 llamadas de coste $T(N/2)$.

Finalmente, se tiene un ciclo de N iteraciones cuyas operaciones en cada caso son de orden constante, produciendo un coste de $\mathcal{O}(N)$. De esta forma, agrupando estos resultados parciales, se puede escribir la ecuación de recurrencia para este algoritmo:

$$T(N) = \mathcal{O}(1) + \mathcal{O}(N/2) + 2T(N/2) + \mathcal{O}(N)$$

$$T(N) = 1 + N + 2T(N/2)$$

$$T(N) = 2T(N/2) + N$$

Como se puede ver, es posible aplicar el teorema maestro, definiendo:

$$a = 2 \geq 1$$

$$b = 2 > 1$$

$$f(N) = N$$

Utilizando el segundo caso del teorema:

$$\exists k \geq 0 \quad / \quad N \in \Theta(N^{\log_b(a)} \log^k(N))$$

$$\Rightarrow T(N) \in \Theta(N^{\log_b(a)} \log^{k+1}(N))$$

Es fácil ver que con $k = 0$ dicha condición se cumple, por lo tanto el resultado final es:

$$T(N) \in \Theta(N \log N)$$

Este resultado es coherente, ya que el algoritmo utiliza la técnica de "divide y vencerás" la recurrencia es análoga al caso del conocido *MergeSort*.

6.2. DFT

6.2.1. Complejidad Temporal

Para estudiar el costo temporal de esta implementación — $T(N)$ — se analizó cada línea de código de la función *calculate_dft_generic()*.

Al principio, todas las sentencias son de orden constante hasta que aparece el primer ciclo de N iteraciones. Dentro de este hay otro ciclo de N iteraciones y 2 sentencias de orden constante, mientras que en el ciclo anidado hay una llamada a una función recursiva (*pow_Complex*):

```

1 static Vector<Complex>
2 calculate_dft_generic(Vector<Complex> const &x, bool inverse)
3 {

```

```

4   Complex aux;
5   size_t N;
6
7   N = x.size();
8
9   // Por defecto se calcula la DFT con estos parámetros:
10  double factor = 1;
11  int W_phase_sign = -1;
12
13  // En caso de tener que calcular la inversa,
14  // modifico el factor de escala y el signo de la fase de W.
15  if (inverse)
16  {
17      factor = 1.0/N;
18      W_phase_sign = 1;
19  }
20
21  Vector<Complex> X(N);
22
23  Complex W(cos((2*PI)/N),
24            W_phase_sign*sin((2*PI)/N));
25
26  for(size_t k=0;k<N;k++) {
27      for(size_t n=0;n<N;n++) {
28          aux += x[n] * pow_complex(W, n*k);
29      }
30      X[k] = factor * aux;
31      aux = 0;
32  }
33  return X;
34 }

```

Analizamos el coste temporal $T_p(p)$ de la función (*pow_Complex*):

```

1   Complex
2   pow_complex(Complex const &z, size_t p)
3   {
4       if(!p) return 1;
5
6       if(p == 1){
7           return z;
8       }
9       else{
10          Complex aux = pow_complex(z, p/2);
11          if(!(p%2))
12              return aux * aux;
13          else
14              return aux * aux * z;
15      }
16  }

```

Se observa que todas las operaciones son de orden constante $\mathcal{O}(1)$ y a continuación se tiene una llamada recursiva. Dado que el tamaño del problema se reduce a la mitad, tenemos 1 llamada de coste $T_p(p/2)$. Agrupando estos resultados, se puede escribir la ecuación de recurrencia para este algoritmo:

$$T_p(p) = \mathcal{O}(1) + T_p(p/2)$$

$$T_p(p) = T_p(p/2) + 1$$

Como se puede ver, es posible aplicar el teorema maestro, definiendo:

$$\begin{aligned} a &= 1 \geq 1 \\ b &= 2 > 1 \\ f(p) &= 1 \end{aligned}$$

Utilizando el segundo caso del teorema:

$$\begin{aligned} \exists k \geq 0 \quad / \quad f(p) &\in \Theta(p^{\log_b(a)} \log^k(p)) \\ \Rightarrow T_p(p) &\in \Theta(p^{\log_b(a)} \log^{k+1}(p)) \end{aligned}$$

Es fácil ver que con $k = 0$ dicha condición se cumple, por lo tanto el resultado final es:

$$T_p(p) \in \Theta(\log p)$$

Una vez sabido el coste temporal de este algoritmo podemos calcular el de la función principal. Como se había planteado anteriormente, consta de 2 ciclos anidados de N iteraciones. El coste del segundo ciclo está dado por:

$$\begin{aligned} T(N) &= (\mathcal{O}(1) + \Theta(\log N)) * N \\ \Rightarrow T(N) &\in \Theta(N \log N) \end{aligned}$$

Entonces el coste total del primer ciclo es:

$$\begin{aligned} T(N) &= (\mathcal{O}(1) + \Theta(N \log N)) * N \\ \Rightarrow T(N) &\in \Theta(N^2 \log N) \end{aligned}$$

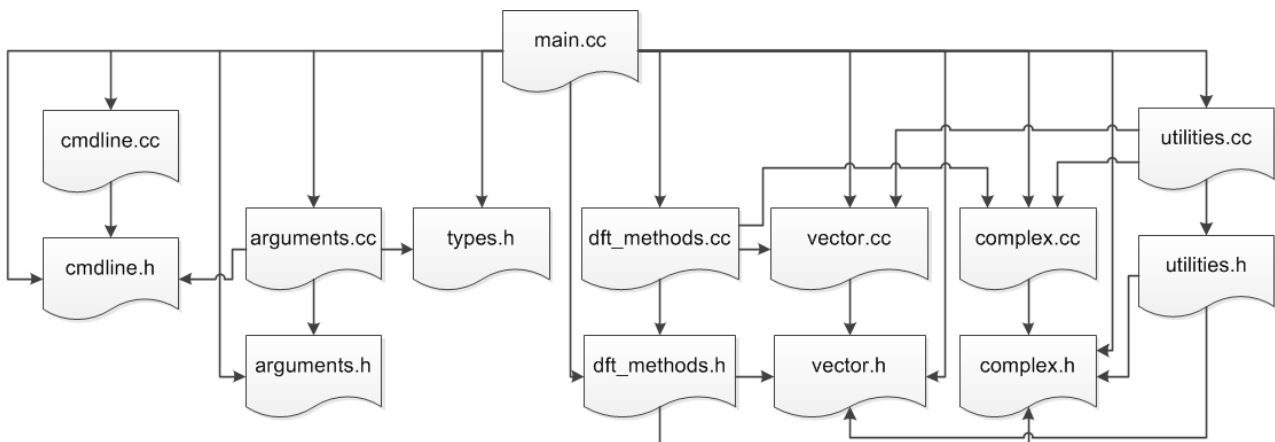
Juntando todos los resultados parciales tenemos que el coste total del algoritmo es:

$$\begin{aligned} T(N) &= \mathcal{O}(1) + \Theta(N^2 \log N) \\ \Rightarrow T(N) &\in \Theta(N^2 \log N) \end{aligned}$$

En conclusión se puede ver que si la función *pow.Complex()* fuera reemplazada por una expresión de orden constante (como por ejemplo la creación del número complejo W directamente en cada iteración, como se hizo en la implementación de la FFT), entonces se perdería la componente logarítmica de la complejidad, quedando el resultado final:

$$T(N) \in \Theta(N^2)$$

7. Estructura de archivos



8. Compilación

Para construir la aplicación, se utilizó el compilador *g++*, de la Free Software Foundation ¹, con las opciones `-Wall` y `-pedantic`, que activan cualquier tipo de advertencia además de los errores de compilación y se restringen a ISO C++.

El proceso de compilación se realiza con el comando `make` que ejecuta el archivo "makefile", el cual se muestra a continuación:

```

1 CC      = g++
2 CCFLAGS = -g -Wall -pedantic
3 LDFLAGS =
4 OBJS = main.o complex.o cmdline.o arguments.o dft_methods.o utilities.o
5 PROGRAMNAME = tp1
6
7 all: tp1
8 #      @/bin/true
9
10 tp1: $(OBJS)
11      $(CC) $(LDFLAGS) $(OBJS) -o $(PROGRAMNAME)
12
13 main.o: main.cc cmdline.h arguments.h complex.h vector.h dft_methods.h utilities
14      .h types.h
15      $(CC) $(CCFLAGS) -c main.cc
16
17 complex.o: complex.cc complex.h
18      $(CC) $(CCFLAGS) -c complex.cc
19
20 cmdline.o: cmdline.cc cmdline.h
21      $(CC) $(CCFLAGS) -c cmdline.cc
22
23 arguments.o: arguments.cc arguments.h cmdline.h types.h
24      $(CC) $(CCFLAGS) -c arguments.cc
25
26 dft_methods.o: dft_methods.cc dft_methods.h complex.h vector.h utilities.h
27      $(CC) $(CCFLAGS) -c dft_methods.cc
28
29 utilities.o: utilities.cc utilities.h complex.h vector.h
30      $(CC) $(CCFLAGS) -c utilities.cc
31
32 clean:
33      rm *.o

```

9. Casos de prueba

Se realizó un *script* para la ejecución de todos los casos de prueba.

```

1 #!/bin/bash
2
3 # Script de tests automáticos para tp1.
4
5 echo Casos de prueba según la especificación del TP
6 echo
7 echo Caso 1
8 echo "$ cat entrada.txt"
9 cat entrada.txt

```

¹www.fsf.org

```

10 echo "$ ./tp1 < entrada.txt"
11 ./tp1 < entrada.txt
12 echo "$ ./tp1 -m fft < entrada.txt"
13 ./tp1 -m fft < entrada.txt
14 echo "$ ./tp1 -m dft < entrada.txt"
15 ./tp1 -m dft < entrada.txt
16 echo
17
18 echo Caso 2
19 echo "$ cat entrada2.txt"
20 cat entrada2.txt
21 echo "$ ./tp1 < entrada2.txt"
22 ./tp1 < entrada2.txt
23 echo "$ ./tp1 -m fft < entrada2.txt"
24 ./tp1 -m fft < entrada2.txt
25 echo "$ ./tp1 -m dft < entrada2.txt"
26 ./tp1 -m dft < entrada2.txt
27 echo
28
29 echo Caso 4
30 echo "$ cat entrada4.txt"
31 cat entrada4.txt
32 echo "$ ./tp1 -m ifft < entrada4.txt"
33 ./tp1 -m ifft < entrada4.txt
34 echo "$ ./tp1 -m idft -o salida4.txt < entrada4.txt"
35 ./tp1 -m idft -o salida4.txt < entrada4.txt
36 echo "$ cat salida4.txt"
37 cat salida4.txt
38 echo

```

A continuación, se muestra el resultado de la ejecución de dicho *script*.

9.1. Caso 1

```

Caso 1
$ cat entrada.txt
1 1 1 1
$ ./tp1 < entrada.txt
(4, 0)
(-1.22461e-16, -1.22461e-16)
(0, -2.44921e-16)
(1.22461e-16, -1.22461e-16)
$ ./tp1 -m fft < entrada.txt
(4, 0)
(-1.22461e-16, -1.22461e-16)
(0, -2.44921e-16)
(1.22461e-16, -1.22461e-16)
$ ./tp1 -m dft < entrada.txt
(4, 0)
(-1.83691e-16, -2.22045e-16)
(0, -2.44921e-16)
(3.29028e-16, -3.33067e-16)

```

9.2. Caso 2

```
Caso 2
$ cat entrada2.txt
1 0 0 0 0 0 0 0
$ ./tp1 < entrada2.txt
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
$ ./tp1 -m fft < entrada2.txt
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
$ ./tp1 -m dft < entrada2.txt
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
(1, 0)
```

9.3. Caso 3

9.4. Caso 4

```
Caso 4
$ cat entrada4.txt
0 0 4 0
$ ./tp1 -m ifft < entrada4.txt
(1, 0)
(-1, -1.22461e-16)
(1, 0)
(-1, -1.22461e-16)
$ ./tp1 -m idft -o salida4.txt < entrada4.txt
$ cat salida4.txt
(1, 0)
(-1, 1.22461e-16)
(1, -2.44921e-16)
(-1, 3.67382e-16)
```

10. Código

10.1. Programa principal

10.1.1. main.cc

```

1  // ----- //
2  // Facultad de Ingeniería de la Universidad de Buenos Aires
3  // Algoritmos y Programación II
4  // 1er Cuatrimestre de 2015
5  // Trabajo Práctico 1: Recursividad
6  // Cálculo de DFT
7  //
8  // main.cc
9  // Archivo principal donde se ejecuta el main.
10 // ----- //
11
12 #include <iostream>
13 #include <cstdlib>
14 #include <sstream>
15
16 #include "cmdline.h"
17 #include "arguments.h"
18 #include "complex.h"
19 #include "vector.h"
20 // En definitiva lo que calculamos en todos los casos es la DFT
21 // (incluso en el caso de la fft, es un algoritmo para calcular la DFT)
22 #include "dft_methods.h"
23 #include "utilities.h"
24 #include "types.h"
25
26 using namespace std;
27
28
29 // Coleccion de funciones para transformar con los distintos métodos
30 Vector<Complex> (*transform []) (Vector<Complex> const &) = {
31
32     calculate_dft ,
33     calculate_idft ,
34     calculate_fft ,
35     calculate_ifft
36     calculate_fft_iter ,
37     calculate_ifft_iter
38
39 };
40
41 extern option_t options [];
42 extern istream *iss;
43 extern ostream *oss;
44 extern method_option_t method_option;
45
46 int main(int argc , char *argv [])
47 {
48     Complex input_complex;
49     Vector<Complex> input;
50     Vector<Complex> output;

```

```

51
52 // Parsear argumentos de invocacion
53 cmdline cmdl(options);
54 cmdl.parse(argc, argv);
55
56 // Mientras haya complejos en la entrada
57 // Cargar vector de entrada
58 while(*iss >> input_complex)
59 {
60     input.push_back(input_complex);
61 }
62
63 // Si el tamaño de la entrada no es potencia de 2 se completa con ceros
64 // hasta llevarla a la potencia de 2 más cercana (Zero-Padding)
65 set_up_input(input);
66
67 // Transformar
68 output = (transform[method_option])(input);
69
70 // Imprimir por la salida especificada por el usuario
71 for(int i=0; i<output.size(); i++)
72 {
73     *oss << output[i] << endl;
74 }
75
76
77 return EXIT_SUCCESS;
78
79 }

```

10.2. Parseo de opciones del programa

10.2.1. types.h

```

1 // ----- //
2 // Facultad de Ingeniería de la Universidad de Buenos Aires
3 // Algoritmos y Programación II
4 // 1° Cuatrimestre de 2015
5 // Trabajo Práctico 1: Recursividad
6 // Cálculo de DFT
7 //
8 // types.h
9 // Tipos definidos para el proposito de la aplicacion
10 // - Opciones posibles de métodos de transformada.
11 // ----- //
12
13 #ifndef _TYPES_H_INCLUDED_
14 #define _TYPES_H_INCLUDED_
15
16 #include <iostream>
17
18 typedef enum{
19
20     METHOD_OPTION_DFT = 0,
21     METHOD_OPTION_IDFT = 1,
22     METHOD_OPTION_FFT = 2,
23     METHOD_OPTION_IFFT = 3,

```

```

24     METHOD.OPTION.FFT.ITER = 4,
25     METHOD.OPTION.IFFT.ITER = 5
26
27 } method_option_t;
28
29 #endif

```

10.2.2. 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 funcin 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 public:
39     cmdline(option_t *);
40     void parse(int, char * const []);
41 };
42
43 #endif

```

10.2.3. cmdline.cc

```

1  // cmdline - procesamiento de opciones en la lnea de comando.
2  //

```

```

3 // $Date: 2012/09/14 13:08:33 $
4 //
5 #include <string>
6 #include <cstdlib>
7 #include <iostream>
8
9 #include "cmdline.h"
10
11 using namespace std;
12
13
14
15 cmdline::cmdline()
16 {
17 }
18
19 cmdline::cmdline(option_t *table) : option_table(table)
20 {
21     /*
22     - Lo mismo que hacer:
23
24     option_table = table;
25
26     Siendo "option_table" un atributo de la clase cmdline
27     y table un puntero a objeto o struct de "option_t".
28
29     Se estara contruyendo una instancia de la clase cmdline
30     cargandole los datos que se hayan en table (la table con
31     las opciones, ver el cdigo en main.cc)
32
33     */
34 }
35
36 void
37 cmdline::parse(int argc, char * const argv[])
38 {
39 #define END_OF_OPTIONS(p) \
40     ((p)->short_name == 0 \
41     && (p)->long_name == 0 \
42     && (p)->parse == 0)
43
44 // Primer pasada por la secuencia de opciones: marcamos
45 // todas las opciones, como no procesadas. Ver cdigo de
46 // abajo.
47 //
48 for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op)
49     op->flags &= ~OPT_SEEN;
50
51 // Recorremos el arreglo argv. En cada paso, vemos
52 // si se trata de una opcin corta, o larga. Luego,
53 // llamamos a la funcin de parseo correspondiente.
54 //
55 for (int i = 1; i < argc; ++i) {
56     // Todos los parmetros de este programa deben
57     // pasarse en forma de opciones. Encontrar un
58     // parametro no-opcin es un error.

```

```

59 //
60 if (argv[i][0] != '-') {
61     cerr << "Invalid non-option argument: "
62         << argv[i]
63         << endl;
64     exit(1);
65 }
66
67 // Usamos "--" para marcar el fin de las
68 // opciones; todo los argumentos que puedan
69 // estar a continuacin no son interpretados
70 // como opciones.
71 //
72 if (argv[i][1] == '-'
73     && argv[i][2] == 0)
74     break;
75
76 // Finalmente, vemos si se trata o no de una
77 // opción larga; y llamamos al mtodo que se
78 // encarga de cada caso.
79 //
80 if (argv[i][1] == '-')
81     i += do_long_opt(&argv[i][2], argv[i + 1]);
82 else
83     i += do_short_opt(&argv[i][1], argv[i + 1]);
84 }
85
86 // Segunda pasada: procesamos aquellas opciones que,
87 // (1) no hayan figurado explícitamente en la línea
88 // de comandos, y (2) tengan valor por defecto.
89 //
90 for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op) {
91 #define OPTION_NAME(op) \
92     (op->short_name ? op->short_name : op->long_name)
93     if (op->flags & OPT_SEEN)
94         continue;
95     if (op->flags & OPT_MANDATORY) {
96         cerr << "Option "
97             << "_"
98             << OPTION_NAME(op)
99             << " is mandatory."
100             << "\n";
101         exit(1);
102     }
103     if (op->def_value == 0)
104         continue;
105     op->parse(string(op->def_value));
106 }
107 }
108
109 int
110 cmdline::do_long_opt(const char *opt, const char *arg)
111 {
112     // Recorremos la tabla de opciones, y buscamos la
113     // entrada larga que se corresponda con la opción de
114     // línea de comandos. De no encontrarse, indicamos el

```



```

115 // error.
116 //
117 for (option_t *op = option_table; op->long_name != 0; ++op) {
118     if (string(opt) == string(op->long_name)) {
119         // Marcamos esta opcion como usada en
120         // forma explícita, para evitar tener
121         // que inicializarla con el valor por
122         // defecto.
123         //
124         op->flags |= OPT_SEEN;
125
126         if (op->has_arg) {
127             // Como se trata de una opcion
128             // con argumento, verificamos que
129             // el mismo haya sido provisto.
130             //
131             if (arg == 0) {
132                 cerr << "Option requires argument: "
133                     << "___"
134                     << opt
135                     << "\n";
136                 exit(1);
137             }
138             op->parse(string(arg));
139             return 1;
140         } else {
141             // Opcion sin argumento.
142             //
143             op->parse(string(""));
144             return 0;
145         }
146     }
147 }
148
149 // Error: opcion no reconocida. Imprimimos un mensaje
150 // de error, y finalizamos la ejecucion del programa.
151 //
152 cerr << "Unknown option: "
153     << "___"
154     << opt
155     << ".\n";
156 exit(1);
157
158 // Algunos compiladores se quejan con funciones que
159 // lógicamente no pueden terminar, y que no devuelven
160 // un valor en esta última parte.
161 //
162 return -1;
163 }
164
165
166 int
167 cmdline::do_short_opt(const char *opt, const char *arg)
168 {
169     option_t *op;
170

```

```

171 // Recorremos la tabla de opciones, y buscamos la
172 // entrada corta que se corresponda con la opcin de
173 // lnea de comandos. De no encontrarse, indicamos el
174 // error.
175 //
176 for (op = option_table; op->short_name != 0; ++op) {
177     if (string(opt) == string(op->short_name)) {
178         // Marcamos esta opcin como usada en
179         // forma explcita, para evitar tener
180         // que inicializarla con el valor por
181         // defecto.
182         //
183         op->flags |= OPT_SEEN;
184
185         if (op->has_arg) {
186             // Como se trata de una opcin
187             // con argumento, verificamos que
188             // el mismo haya sido provisto.
189             //
190             if (arg == 0) {
191                 cerr << "Option requires argument: "
192                     << "_"
193                     << opt
194                     << "\n";
195                 exit(1);
196             }
197             op->parse(string(arg));
198             return 1;
199         } else {
200             // Opcin sin argumento.
201             //
202             op->parse(string(""));
203             return 0;
204         }
205     }
206 }
207
208 // Error: opcin no reconocida. Imprimimos un mensaje
209 // de error, y finalizamos la ejecucin del programa.
210 //
211 cerr << "Unknown option: "
212     << "_"
213     << opt
214     << ". "
215     << endl;
216 exit(1);
217
218 // Algunos compiladores se quejan con funciones que
219 // lgicamente no pueden terminar, y que no devuelven
220 // un valor en esta ltima parte.
221 //
222 return -1;
223 }

```

10.2.4. arguments.h

1 // _____ //

```

2 // Facultad de Ingeniería de la Universidad de Buenos Aires
3 // Algoritmos y Programación II
4 // 1er Cuatrimestre de 2015
5 // Trabajo Práctico 1: Recursividad
6 // Cálculo de DFT
7 //
8 // arguments.h
9 // Funciones a llamar para cada opcion posible de la aplicacion
10 // Nombres de los argumentos de la opcion "--format"
11 // ----- //
12
13 #ifndef _ARGUMENTS_H_INCLUDED_
14 #define _ARGUMENTS_H_INCLUDED_
15
16 #include <iostream>
17
18 #define METHOD_OPTIONS 6
19 #define METHOD_DFT "dft"
20 #define METHOD_IDFT "idft"
21 #define METHOD_FFT "fft"
22 #define METHOD_IFFT "ifft"
23 #define METHOD_FFT_ITER "fft-iter"
24 #define METHOD_IFFT_ITER "ifft-iter"
25
26 void opt_input(std::string const &);
27 void opt_output(std::string const &);
28 void opt_method(std::string const &);
29
30 #endif

```

10.2.5. arguments.cc

```

1 // ----- //
2 // Facultad de Ingeniería de la Universidad de Buenos Aires
3 // Algoritmos y Programación II
4 // 1er Cuatrimestre de 2015
5 // Trabajo Práctico 1: Recursividad
6 // Cálculo de DFT
7 //
8 // arguments.cc
9 // Funciones a llamar para cada opción posible de la aplicación
10 // ----- //
11
12 #include <iostream>
13 #include <fstream>
14 #include <sstream>
15 #include <cstdlib>
16 #include <cstring>
17
18 #include "arguments.h"
19 #include "cmdline.h"
20 #include "types.h"
21
22 using namespace std;
23
24
25 // Opciones de argumentos de invocacion

```

```

26 option_t options [] = {
27     {1, "i", "input", "--", opt_input, OPT_SEEN},
28     {1, "o", "output", "--", opt_output, OPT_SEEN},
29     {1, "m", "method", "fft", opt_method, OPT_SEEN},
30     {0, },
31 };
32
33 // Nombres de los argumentos de la opcion "--method"
34 string description_method_option [] = {
35     METHOD_DFT,
36     METHOD_IDFT,
37     METHOD_FFT,
38     METHOD_IFFT,
39     METHOD_FFT_ITER,
40     METHOD_IFFT_ITER
41 };
42
43
44 istream *iss;
45 ostream *oss;
46 fstream ifs;
47 fstream ofs;
48 method_option_t method_option;
49
50 void
51 opt_input(string const &arg)
52 {
53     // Si el nombre del archivos es "--", usaremos la entrada
54     // estándar. De lo contrario, abrimos un archivo en modo
55     // de lectura.
56     //
57     if (arg == "--") {
58         iss = &cin; // Establezco la entrada estandar cin como flujo de entrada
59     }
60     else {
61         ifs.open(arg.c_str(), ios::in); // c_str(): Returns a pointer to an array
62         // that contains a null-terminated
63         // sequence of characters (i.e., a C-string) representing
64         // the current value of the string object.
65         iss = &ifs;
66     }
67     // Verificamos que el stream este OK.
68     //
69     if (!iss->good()) {
70         cerr << "Cannot open "
71             << arg
72             << ". "
73             << endl;
74         exit(1);
75     }
76 }
77
78 void
79 opt_output(string const &arg)
80 {

```

```

81 // Si el nombre del archivos es "-", usaremos la salida
82 // estándar. De lo contrario, abrimos un archivo en modo
83 // de escritura.
84 //
85 if (arg == "-") {
86     oss = &cout; // Establezco la salida estandar cout como flujo de salida
87 } else {
88     ofs.open(arg.c_str(), ios::out);
89     oss = &ofs;
90 }
91
92 // Verificamos que el stream este OK.
93 //
94 if (!oss->good()) {
95     cerr << "Cannot open "
96         << arg
97         << ". "
98         << endl;
99     exit(1); // EXIT: Terminación del programa en su totalidad
100 }
101 }
102
103 void
104 opt_method(string const &arg)
105 {
106     size_t i;
107     // Recorremos diccionario de argumentos hasta encontrar uno que coincida
108     for(i=0; i < METHOD_OPTIONS; i++) {
109         if(arg == description_method_option[i]) {
110             method_option = (method_option_t)i; // Casteo
111             break;
112         }
113     }
114     // Si recorrio todo el diccionario, el argumento no esta implementado
115     if (i == METHOD_OPTIONS) {
116         cerr << "Unknown format" << endl;
117         exit(1);
118     }
119 }

```

10.3. Clase Vector

10.3.1. vector.h

```

1 // ----- //
2 // Facultad de Ingeniería de la Universidad de Buenos Aires
3 // Algoritmos y Programación II
4 // 1° Cuatrimestre de 2015
5 // Trabajo Práctico 1: Recursividad
6 // Cálculo de DFT
7 //
8 // vector.h
9 // Interface de la clase Vector
10 // ----- //
11
12
13 #ifndef _VECTOR_H_

```

```

14 #define _VECTOR_H_
15
16 #include <iostream>
17 #include <cstdlib>
18 using namespace std;
19
20
21 template <typename T>
22 class Vector
23 {
24     public:
25         Vector();
26         Vector(int);
27         Vector(const Vector<T> &);
28         ~Vector();
29
30         int size() const;
31         const Vector<T> &operator=(const Vector<T> &); // operador asignacion
32         const T &operator [] (int) const;
33         T &operator [] (int);
34
35         void push_back(const T &); // Alta al final
36         void pop_back(); // Baja al final
37
38
39     private:
40         int size_;
41         T *ptr_;
42 };
43
44 // Se incluye el .cc que contiene la implementación para que pueda
45 // compilar bien y se mantenga la separación de interface-implementación
46 // en 2 archivos distintos. Esto se debe al uso de plantillas.
47 #include "vector.cc"
48
49 #endif // _VECTOR_H_

```

10.3.2. vector.cc

```

1 // ----- //
2 // Facultad de Ingeniería de la Universidad de Buenos Aires
3 // Algoritmos y Programación II
4 // 1° Cuatrimestre de 2015
5 // Trabajo Práctico 1: Recursividad
6 // Cálculo de DFT
7 //
8 // vector.cc
9 // Implementación de la clase Vector
10 // ----- //
11
12 // #include "vector.h"
13
14 // Constructor por defecto
15 template <typename T>
16 Vector<T>::Vector()
17 {
18     ptr_ = NULL;

```

```
19     size_ = 0;
20 }
21
22 // Constructor con argumento
23 template <typename T>
24 Vector<T>::Vector(int s)
25 {
26     if (s <= 0)
27     {
28         exit(1);
29     }
30     else
31     {
32         size_ = s;
33         ptr_ = new T[size_];
34     }
35 }
36
37 // Constructor por copia
38 template <class T>
39 Vector<T>::Vector(const Vector<T> &v)
40 {
41     size_ = v.size_ ;
42     ptr_ = new T[size_];
43     for (int i=0; i < size_ ; i++)
44         ptr_[i] = v.ptr_[i];
45 }
46
47
48 template <typename T>
49 Vector<T>::~~Vector()
50 {
51     if (ptr_)
52         delete [] ptr_ ;
53 }
54
55
56 template <typename T>
57 int
58 Vector<T>::size() const
59 {
60     return size_ ;
61 }
62
63
64 // Operador asignacion
65 template <typename T>
66 const Vector<T> &
67 Vector<T>::operator=(const Vector<T> &rigth)
68 {
69     if (this != &rigth)
70     {
71         if (size_ != rigth.size_)
72         {
73             T *aux;
74             aux = new T[rigth.size_];
```

```

75         delete [] ptr_; // si llegó acá es que obtuvo el espacio; libera el
              anterior espacio
76
77         size_ = righth.size_;
78         ptr_ = aux;
79         for (int i=0; i < size_; i++)
80             ptr_[i] = righth.ptr_[i];
81
82         return *this;
83     }
84     else
85     {
86         for (int i=0; i < size_; i++)
87             ptr_[i] = righth.ptr_[i];
88
89         return *this;
90     }
91 }
92 return *this;
93 }
94
95
96 // Operador Sub-índice que devuelve rvalue constante
97 // En caso de pasar un sub-índice fuera de rango el programa finaliza con error
98 template <typename T>
99 const T &
100 Vector<T>::operator [] (int i) const
101 {
102     if (i < 0 || i >= size_)
103         exit(1);
104
105     return ptr_[i];
106 }
107
108 // Operador Sub-índice que devuelve lvalue modificable
109 // En caso de pasar un sub-índice fuera de rango el programa finaliza con error
110 template <typename T>
111 T &
112 Vector<T>::operator [] (int i)
113 {
114     if (i < 0 || i >= size_)
115         exit(1);
116
117     return ptr_[i];
118 }
119
120
121 // Alta al final
122 template <typename T>
123 void
124 Vector<T>::push_back(const T & data)
125 {
126     T *aux;
127     aux = new T[size_ + 1];
128     for(int i=0; i<size_; i++)

```



```

130     aux[i] = ptr_[i];
131     aux[size_] = data;
132
133     delete [] ptr_;
134     ptr_ = aux;
135     size_++;
136 }
137
138
139
140 // Baja al final
141 template <typename T>
142 void
143 Vector<T>::pop_back()
144 {
145     T *aux;
146     aux = new T[size_ - 1];
147     for(int i=0; i<(size_-1); i++)
148         aux[i] = ptr_[i];
149
150     delete [] ptr_;
151     ptr_ = aux;
152     size_--;
153 }

```

10.4. Clase Complejo

10.4.1. complex.h

```

1  #ifndef _COMPLEX_H_INCLUDED_
2  #define _COMPLEX_H_INCLUDED_
3
4  #include <iostream>
5
6  class Complex
7  {
8  public:
9      Complex();
10     Complex(double);
11     Complex(double, double);
12     Complex(const Complex &);
13     Complex const &operator=(Complex const &);
14     Complex const &operator*=(Complex const &);
15     Complex const &operator+=(Complex const &);
16     Complex const &operator-=(Complex const &);
17     ~Complex();
18
19     double real() const;
20     double imag() const;
21     double abs() const;
22     double abs2() const;
23     double phase() const;
24     Complex const &conjugate();
25     Complex const conjugated() const;
26     bool iszero() const;
27
28     friend Complex const operator+(Complex const &, Complex const &);

```

```

29     friend Complex const operator-(Complex const &, Complex const &);
30     friend Complex const operator*(Complex const &, Complex const &);
31     friend Complex const operator/(Complex const &, Complex const &);
32     friend Complex const operator/(Complex const &, double);
33
34     friend bool operator==(Complex const &, double);
35     friend bool operator==(Complex const &, Complex const &);
36
37     friend std::ostream &operator<<(std::ostream &, const Complex &);
38     friend std::istream &operator>>(std::istream &, Complex &);
39
40 private:
41     double real_, imag_;
42 }; // class Complex
43
44 #endif // _COMPLEX_H_INCLUDED_

```

10.4.2. complex.cc

```

1  #include <iostream>
2  #include <cmath>
3
4  #include "complex.h"
5
6  using namespace std;
7
8
9
10 Complex::Complex() : real_(0), imag_(0)
11 {
12 }
13
14 Complex::Complex(double r) : real_(r), imag_(0)
15 {
16 }
17
18 Complex::Complex(double r, double i) : real_(r), imag_(i)
19 {
20 }
21
22 Complex::Complex(Complex const &c) : real_(c.real_), imag_(c.imag_)
23 {
24 }
25
26 Complex const &
27 Complex::operator=(Complex const &c)
28 {
29     real_ = c.real_;
30     imag_ = c.imag_;
31     return *this;
32 }
33
34 Complex const &
35 Complex::operator*=(Complex const &c)
36 {
37     double re = real_ * c.real_
38             - imag_ * c.imag_;

```

```
39     double im = real_ * c.imag_
40           + imag_ * c.real_;
41     real_ = re, imag_ = im;
42     return *this;
43 }
44
45 Complex const &
46 Complex::operator+=(Complex const &c)
47 {
48     double re = real_ + c.real_;
49     double im = imag_ + c.imag_;
50     real_ = re, imag_ = im;
51     return *this;
52 }
53
54 Complex const &
55 Complex::operator-=(Complex const &c)
56 {
57     double re = real_ - c.real_;
58     double im = imag_ - c.imag_;
59     real_ = re, imag_ = im;
60     return *this;
61 }
62
63 Complex::~~Complex()
64 {
65 }
66
67 double
68 Complex::real() const
69 {
70     return real_;
71 }
72
73 double Complex::imag() const
74 {
75     return imag_;
76 }
77
78 double
79 Complex::abs() const
80 {
81     return std::sqrt(real_ * real_ + imag_ * imag_);
82 }
83
84 double
85 Complex::abs2() const
86 {
87     return real_ * real_ + imag_ * imag_;
88 }
89
90 double
91 Complex::phase() const
92 {
93     return atan2(imag_, real_);
94 }
```

```

95
96 Complex const &
97 Complex::conjugate()
98 {
99     imag_*= -1;
100     return *this;
101 }
102
103 Complex const
104 Complex::conjugated() const
105 {
106     return Complex(real_ , -imag_);
107 }
108
109 bool
110 Complex::iszero() const
111 {
112     #define ZERO(x) ((x) == +0.0 && (x) == -0.0)
113     return ZERO(real_) && ZERO(imag_) ? true : false;
114 }
115
116 Complex const
117 operator+(Complex const &x, Complex const &y)
118 {
119     Complex z(x.real_ + y.real_ , x.imag_ + y.imag_);
120     return z;
121 }
122
123 Complex const
124 operator-(Complex const &x, Complex const &y)
125 {
126     Complex r(x.real_ - y.real_ , x.imag_ - y.imag_);
127     return r;
128 }
129
130 Complex const
131 operator*(Complex const &x, Complex const &y)
132 {
133     Complex r(x.real_ * y.real_ - x.imag_ * y.imag_ ,
134             x.real_ * y.imag_ + x.imag_ * y.real_);
135     return r;
136 }
137
138 Complex const
139 operator/(Complex const &x, Complex const &y)
140 {
141     return x * y.conjugated() / y.abs2();
142 }
143
144 Complex const
145 operator/(Complex const &c, double f)
146 {
147     return Complex(c.real_ / f, c.imag_ / f);
148 }
149
150 bool

```

```

151 operator==(Complex const &c, double f)
152 {
153     bool b = (c.imag_ != 0 || c.real_ != f) ? false : true;
154     return b;
155 }
156
157 bool
158 operator==(Complex const &x, Complex const &y)
159 {
160     bool b = (x.real_ != y.real_ || x.imag_ != y.imag_) ? false : true;
161     return b;
162 }
163
164 ostream &
165 operator<<(ostream &os, const Complex &c)
166 {
167     return os << "("
168             << c.real_
169             << ", "
170             << c.imag_
171             << ")";
172 }
173
174 istream &
175 operator>>(istream &is, Complex &c)
176 {
177     int good = false;
178     int bad = false;
179     double re = 0;
180     double im = 0;
181     char ch = 0;
182
183     if (is >> ch
184         && ch == '(') {
185         if (is >> re
186             && is >> ch
187             && ch == ', '
188             && is >> im
189             && is >> ch
190             && ch == ')')
191             good = true;
192         else
193             bad = true;
194     } else if (is.good()) {
195         is.putback(ch);
196         if (is >> re)
197             good = true;
198         else
199             bad = true;
200     }
201
202     if (good)
203         c.real_ = re, c.imag_ = im;
204     if (bad)
205         is.clear(ios::badbit);
206

```

```

207     return is;
208 }

```

10.5. Funciones utilitarias

10.5.1. utilities.h

```

1  // ----- //
2  // Facultad de Ingeniería de la Universidad de Buenos Aires
3  // Algoritmos y Programación II
4  // 1° Cuatrimestre de 2015
5  // Trabajo Práctico 1: Recursividad
6  // Cálculo de DFT
7  //
8  // utilities.h
9  // Funciones utilitarias para el propósito de la aplicación
10 // ----- //
11
12 #ifndef _UTILITIES_H_INCLUDED_
13 #define _UTILITIES_H_INCLUDED_
14
15 #include <iostream>
16
17 #include "complex.h"
18 #include "vector.h"
19
20 void set_up_input(Vector<Complex> &);
21 Complex pow_complex(Complex const &, size_t);
22 size_t my_pow(size_t const &, size_t);
23
24 #endif

```

10.5.2. utilities.cc

```

1  // ----- //
2  // Facultad de Ingeniería de la Universidad de Buenos Aires
3  // Algoritmos y Programación II
4  // 1° Cuatrimestre de 2015
5  // Trabajo Práctico 1: Recursividad
6  // Cálculo de DFT
7  //
8  // utilities.cc
9  // Funciones utilitarias para el propósito de la aplicación
10 // ----- //
11
12 #include <iostream>
13 #include <fstream>
14 #include <sstream>
15 #include <cmath>
16
17 #include "utilities.h"
18 #include "complex.h"
19
20 using namespace std;
21
22
23

```

```

24 void
25 set_up_input (Vector<Complex> &input)
26 {
27     size_t n = input.size();
28
29     if (log2(n) - (int)log2(n)) {
30         size_t l = (int)log2(n) + 1;
31         size_t last = my_pow(2, l) - input.size();
32         for(size_t i=0; i<last; i++) {
33             input.push_back(0);
34         }
35     }
36     return;
37 }
38
39 Complex
40 pow_complex(Complex const &z, size_t p)
41 {
42     if(!p) return 1;
43
44     if(p == 1){
45         return z;
46     }
47     else{
48         Complex aux = pow_complex(z, p/2);
49         if(!(p%2))
50             return aux * aux;
51         else
52             return aux * aux * z;
53     }
54 }
55
56 size_t
57 my_pow(size_t const &n, size_t p)
58 {
59     if(!p) return 1;
60
61     if(p == 1){
62         return n;
63     }
64     else{
65         size_t aux = my_pow(n, p/2);
66         if(!(p%2))
67             return aux * aux;
68         else
69             return aux * aux * n;
70     }
71 }

```

10.6. Métodos para Transformar

10.6.1. dft_methods.h

```

1 // _____ //
2 // Facultad de Ingeniería de la Universidad de Buenos Aires
3 // Algoritmos y Programación II
4 // 1° Cuatrimestre de 2015

```

```

5 // Trabajo Práctico 1: Recursividad
6 // Cálculo de DFT
7 //
8 // dft_methods.h
9 // Interfaces de los distintos métodos de la transformada.
10 // ----- //
11
12 #ifndef _DFT_METHODS_H_
13 #define _DFT_METHODS_H_
14
15 #include <iostream>
16
17 #include "complex.h"
18 #include "vector.h"
19
20 #define PI 3.14159265358979323846264338327950
21
22 Vector<Complex> calculate_dft (Vector<Complex> const &);
23 Vector<Complex> calculate_idft (Vector<Complex> const &);
24 Vector<Complex> calculate_fft (Vector<Complex> const &);
25 Vector<Complex> calculate_ifft (Vector<Complex> const &);
26
27 Vector<Complex> calculate_fft_iter (Vector<Complex> const &);
28 Vector<Complex> calculate_ifft_iter (Vector<Complex> const &);
29
30 #endif

```

10.6.2. dft_methods.cc

```

1 // ----- //
2 // Facultad de Ingeniería de la Universidad de Buenos Aires
3 // Algoritmos y Programación II
4 // 1° Cuatrimestre de 2015
5 // Trabajo Práctico 1: Recursividad
6 // Cálculo de DFT
7 //
8 // dft_methods.cc
9 // Implementación de los distintos métodos de la transformada.
10 // ----- //
11
12 #include <iostream>
13 #include <cmath>
14
15 #include "dft_methods.h"
16 #include "complex.h"
17 #include "vector.h"
18 #include "utilities.h"
19
20 using namespace std;
21
22 // ----- DFT -----
23 // Función genérica para calcular DFT o IDFT
24 // Oculta al cliente.
25 // Si el flag "inverse" es "true", se calcula la inversa (IDFT)
26 // Caso contrario, la DFT
27 // Algoritmo iterativo para calcular la DFT
28 // Versión que utiliza función "pow" en cada iteración

```



```

29 static Vector<Complex>
30 calculate_dft_generic(Vector<Complex> const &x, bool inverse)
31 {
32     Complex aux;
33     size_t N;
34
35     N = x.size();
36
37     // Por defecto se calcula la DFT con estos parámetros:
38     double factor = 1;
39     int W_phase_sign = -1;
40
41     // En caso de tener que calcular la inversa,
42     // modifico el factor de escala y el signo de la fase de W.
43     if (inverse)
44     {
45         factor = 1.0/N;
46         W_phase_sign = 1;
47     }
48
49     Vector<Complex> X(N);
50
51     Complex W(cos((2*PI)/N),
52              W_phase_sign*sin((2*PI)/N));
53
54     for(size_t k=0;k<N;k++) {
55         for(size_t n=0;n<N;n++) {
56             aux += x[n] * pow_complex(W, n*k);
57         }
58         X[k] = factor * aux;
59         aux = 0;
60     }
61     return X;
62 }
63
64 // Máscara para la DFT
65 // Llama a la función genérica en modo "directa"
66 Vector<Complex>
67 calculate_dft(Vector<Complex> const &x)
68 {
69     bool inverse = false;
70     return calculate_dft_generic(x, inverse);
71 }
72
73 // Máscara para la IDFT
74 // Llama a la función genérica en modo "inversa"
75 Vector<Complex>
76 calculate_idft(Vector<Complex> const &X)
77 {
78     bool inverse = true;
79     return calculate_dft_generic(X, inverse);
80 }
81
82 // ----- FFT -----
83 // Función genérica para calcular FFT o IFFT
84 // Oculta al cliente.

```

```

85 // Si el flag "inverse" es "true", se calcula la inversa (IFFT)
86 // Caso contrario, la FFT
87 // Algoritmo recursivo para calcular la DFT: FFT
88 static Vector<Complex>
89 calculate_fft_generic(Vector<Complex> const &x, bool inverse)
90 {
91     size_t N;
92     N = x.size();
93
94     Vector<Complex> X(N);
95
96     // Por defecto se calcula la FFT con estos parámetros:
97     double factor = 1;
98     int W_phase_sign = -1;
99
100    // En caso de tener que calcular la inversa,
101    // modifico el factor de escala y el signo de la fase de W.
102    if (inverse)
103    {
104        factor = 1.0/N;
105        W_phase_sign = 1;
106    }
107
108    if (N > 1)
109    {
110        // Divido el problema en 2:
111        // Suponemos que la entrada es par y potencia de 2
112        Vector<Complex> p(N/2);
113        Vector<Complex> q(N/2);
114        Vector<Complex> P(N/2);
115        Vector<Complex> Q(N/2);
116        for (size_t i=0; i<N/2; i++)
117        {
118            p[i] = x[2*i];
119            q[i] = x[2*i+1];
120        }
121
122        P = calculate_fft(p);
123        Q = calculate_fft(q);
124
125        // Combino las soluciones:
126        for (size_t k=0; k<N; k++)
127        {
128            Complex W(cos(k*(2*PI)/N),
129                    W_phase_sign*sin(k*(2*PI)/N));
130            // Para que se repitan los elementos cíclicamente, se utiliza la función
131            // módulo
132            size_t k2 = k % (N/2);
133            X[k] = factor * (P[k2] + W*Q[k2]);
134        }
135    }
136    else
137    {
138        X = x;
139    }

```

```

140
141     return X;
142 }
143
144 // Máscara para la FFT
145 // Llama a la función genérica en modo "directa"
146 Vector<Complex>
147 calculate_fft(Vector<Complex> const &x)
148 {
149     bool inverse = false;
150     return calculate_fft_generic(x, inverse);
151 }
152
153 // Máscara para la IFFT
154 // Llama a la función genérica en modo "inversa"
155 Vector<Complex>
156 calculate_ifft(Vector<Complex> const &X)
157 {
158     bool inverse = true;
159     return calculate_fft_generic(X, inverse);
160 }
161
162 // ----- FFT_iter -----
163 // Función genérica para calcular FFT o IFFT versiones iterativas
164 // Oculta al cliente.
165 // Si el flag "inverse" es "true", se calcula la inversa (IFFT)
166 // Caso contrario, la FFT
167 // Algoritmo recursivo para calcular la DFT: FFT
168 static Vector<Complex>
169 calculate_fft_iter_generic(Vector<Complex> const &x, bool inverse)
170 {
171     size_t N;
172     N = x.size();
173
174     Vector<Complex> X(N);
175
176     // Por defecto se calcula la FFT con estos parámetros:
177     double factor = 1;
178     int W_phase_sign = -1;
179
180     // En caso de tener que calcular la inversa,
181     // modifico el factor de escala y el signo de la fase de W.
182     if (inverse)
183     {
184         factor = 1.0/N;
185         W_phase_sign = 1;
186     }
187
188     if (N > 1)
189     {
190         // Divido el problema en 2:
191         // Suponemos que la entrada es par y potencia de 2
192         Vector<Complex> p(N/2);
193         Vector<Complex> q(N/2);
194         Vector<Complex> P(N/2);
195         Vector<Complex> Q(N/2);

```

```

196     for (size_t i=0; i<N/2; i++)
197     {
198         p[i] = x[2*i];
199         q[i] = x[2*i+1];
200     }
201
202     P = calculate_fft_iter(p);
203     Q = calculate_fft_iter(q);
204
205     // Combino las soluciones:
206     for (size_t k=0; k<N; k++)
207     {
208         Complex W(cos(k*(2*PI)/N),
209                 W_phase_sign*sin(k*(2*PI)/N));
210         // Para que se repitan los elementos cíclicamente, se utiliza la función
211         // módulo
212         size_t k2 = k % (N/2);
213         X[k] = factor * (P[k2] + W*Q[k2]);
214     }
215 }
216 else
217 {
218     X = x;
219 }
220
221 return X;
222 }
223 Vector<Complex>
224 calculate_fft_iter(Vector<Complex> const &x)
225 {
226     bool inverse = false;
227     return calculate_fft_iter_generic(x, inverse);
228 }
229
230 Vector<Complex>
231 calculate_ifft_iter(Vector<Complex> const &X)
232 {
233     bool inverse = true;
234     return calculate_fft_iter_generic(X, inverse);
235 }

```

11. Enunciado