75.04/95.12 Algoritmos y Programación II
Trabajo práctico 1: Recursividad

Universidad de Buenos Aires - FIUBA
Primer cuatrimestre de 2016
`$Date: 2016/05/12 02:19:08 $`

1. Objetivos

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

2. Alcance

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

3. Requisitos

El trabajo deberá será 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 6, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Descripción

En esta sección describiremos las 4 variantes de los algoritmos de transformación que implementaremos en este TP.

4.1. DFT: discrete fourier transform (ida)

Este algoritmo permite calcular la transformación de una secuencia de N puntos $x(0), \ldots, x(N-1)$ dada por la fórmula:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}; k = 0, \ldots, N-1 \quad (1)$$
$$; W_N = e^{-j\frac{2\pi}{N}}$$

Es decir, ante una entrada de N valores complejos, producirá una secuencia de salida de igual tipo y longitud.

4.2. iDFT: inverse discrete fourier transform

En este caso queremos hacer la operación inversa: obtener el vector original, x, a partir del vector transformado X,

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}; n = 0, \ldots, N-1 \quad (2)$$

Podemos ver entonces, que se trata de un cálculo muy similar al descripto en la sección anterior: las únicas diferencias están en un factor $1/N$, y en el signo negativo en el exponente de W.

4.3. FFT: fast fourier transform

Se trata de una variante de la DFT que permite obtener *eficientemente* una secuencia de N números X que verifique la fórmula 1.

Aplicando la técnica recursiva *dividir y conquistar*, y separando el vector de entrada x en dos subvectores p y q, que contienen las componentes de índices pares e impares respectivamente, se puede verificar lo siguiente:

$$X(k) = P(k) + W_N^k Q(k); k = 0, \ldots, N-1$$

En donde P y Q son las transformaciones p y q, y el operador de acceso a estos arreglos trabaja en forma modular (es decir, para cualquier par de enterios k y l, tendremos que $P(k) = P(k + l|P|)$). Si N es potencia entera de 2, podemos aplicar dividir y conquistar una y otra vez, y obtener así el siguiente algoritmo:

```
vector fft(vector x)
{
        if ((N = x.length()) >= 2) {
                p = componentes de x con índice par;
                q = componentes de x con índice impar;

                P = fft(p);
                Q = fft(q);

                for (k = 0; k < N; ++k)
                        X[k] = P[k] + W(k, N) * Q[k];
        } else {
                X = x;
        }

        return X;
}
```

4.4. iFFT: inverse FFT

En este caso, queremos hacer la operación inversa: obtener el vector original, x, a partir del vector transformado X, de manera más eficiente que la sugerida en la ecuación 2.

Por lo visto antes, podemos ver que se trata de un cálculo muy similar al de la FFT: como ya vimos en la sección 4.2, las únicas diferencias están en un factor $1/N$, y en el signo negativo en el exponente de W. Por ende, el proceso de derivar el algoritmo recursivo, que calcula eficientemente la fórmula 2, es análogo al descripto en la sección 4.3.

5. Implementación

Para facilitar el proceso de desarrollo y debug, el programa deberá implementar el algoritmo FFT como también la transformada por definición (DFT).

Asimismo, deberá permitir realizar transformaciones en sentido directo e inverso, las cuales nos servirán para depurar las implementaciones de los algoritmos vistos en las secciones 4.3 y 4.4.

Para configurar el funcionamiento del programa, usaremos parámetros pasados por la línea de comando, como hicimos en el TP anterior. Por defecto, el programa se limitará a leer, con formato, la secuencia de números complejos por el stream de texto de entrada `std::cin`, e imprimirá (también en forma de texto con formato) el resultado de la FFT por `std::cout`.

Si bien existen variantes de la FFT que permiten calculas transformaciones de secuencias de longitud arbitraria, en este TP nos limitaremos a realizar transformaciones de secuencias cuya longitud es potencia entera de 2. Para ello, el programa deberá completar con ceros las entradas que no cumplan con este requisito, llevando así su longitud a la potencia entera de 2 que resulte más cercana.

El formato de los streams de entrada y salida difiere del usado en el trabajo anterior: en este caso, cada línea del archivo de entrada contendrá una secuencia de números complejos a transformar, en la cual los elementos están expresados en forma de par ordenado (i.e., (a, b) siendo a la parte real, y b la parte imaginaria); o como números reales sueltos.

Así, cuando el stream de entrada contenga múltiples líneas, el programa deberá ir leyendo línea a línea, transformando cada de ellas, e imprimiendo la salida.

5.1. Interfaz

Las opciones de línea de comando a implementar en este TP son:

- `-o`, o `--output`, permite colocar la secuencia de números complejos de salida en el archivo pasado como argumento; o por salida estándar -`std::cout`- si el argumento es "`-`".
- `-i`, o `--input`, para controlar el stream de entrada, de forma similar a la opción anterior; es decir, el programa leerá de `std::cin` cuando el valor pasado sea "`-`"..
- `-m`, o `--method` permite especificar el algoritmo de cómputo: `dft` (sección 4.1), `idft` (4.2), `fft` (4.3), `ifft` (4.4), `fft-iter` e `ifft-iter` (9). En este TP, el valor por defecto a utilizar es `fft`.

En todos los casos, el formato de los streams de entrada/salida es el de números complejos con formato, expresados como pares ordenados de la forma `($re, $im)`, siendo `$re` la parte real, y `$im` la parte imaginaria. Asimismo, el programa deberá aceptar números reales sueltos en su entrada.

5.2. Ejemplos

Comencemos invocando al programa con las opciones por defecto, es decir, transformando con FFT los datos de `std::cin` y enviando la salida a `std::cout`:

```
$ cat entrada.txt
1 1 1 1
$ tp1 < entrada.txt
(4, 0) (0, 0) (0, 0) (0, 0)
$ tp1 -m fft < entrada.txt
(4, 0) (0, 0) (0, 0) (0, 0)
$ tp1 -m dft < entrada.txt
(4, 0) (0, 0) (0, 0) (0, 0)
```

Observamos que la entrada también puede ser compleja, y contener múltiples vectores a transformar (cada uno en su propia línea):

```
$ cat entrada2.txt
(1, 0)
(1, 0) (0, 0)
(1, 0) (0, 0) 0 (0, 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) (1, 0) (1, 0)
```

Además, podemos invocar al programa para que realice los cálculos con DFT, y así validar los resultados anteriores:

```
$ tp1 -m dft < entrada2.txt
(1, 0) (1, 0) (1, 0) (1, 0) (1, 0) (1, 0) (1, 0) (1, 0)
```

Similarmente, para anti-transformar:

```
$ cat entrada4.txt
(0, 0) (0, 0) (4, 0) (0, 0)
$ tp1 -m ifft < entrada4.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
$ tp1 -m idft -o salida4.txt < entrada4.txt
$ cat salida4.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
```

5.3. Portabilidad

Es deseable que la implementación desarrollada provea un grado mínimo de portabilidad. Sugerimos verificar nuestros programas en alguna versión reciente de UNIX: BSD o Linux.

6. Informe

El informe deberá incluir:

- Documentación relevante al diseño e implementación del programa.
- Documentación relevante a los algoritmos involucrados en la solución del trabajo, incluyendo los detalles omitidos en la descripción del algoritmo de la sección 4.4.
- El análisis de complejidad espacial y temporal de todos los algoritmos involucrados, incluyendo los presentados en la sección 4.
- 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.

7. Fechas

La última fecha de entrega es el jueves 26/5.

8. Optativo: cálculo *in-place*

Esta técnica permite eliminar el uso de los vectores p, q, P, y Q, presentados en el algoritmo de la sección 4.3. Gracias a esto, es posible ahorrar memoria.

Se trata entonces de cuantificar el ahorro, realizando análisis de complejidad espacial; e implementar esta variante del algoritmo en el programa. Para más información ver [1].

9. Optativo: cálculo iterativo

Es posible eliminar las llamadas recursivas en los algoritmos presentados en las secciones 4.3 y 4.4[1]: buscamos obtener un programa adicional que implemente esta variante iterativa, e incluir en el informe los análisis de complejidades, mediciones, y otros datos cuantitativos que permitan comparar ambas implementaciones objetivamente.

[1] Ver, por ejemplo, los gráficos de flujo se señal presentados en [1].

Referencias

[1] Alan V. Oppenheim, Roland W. Schafer. Discrete-time signal processing, second edition. Sección 9.3.1: In-Place Computations. Ed. Prentice Hall.