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

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

2. Introducción

Explicar un poco que es la FT, la DFT y la 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

Explicar a grandes rasgos como funciona el programa, diagrama en bloques. -¿Leer de la entrada a vector, rellenar con ceros, transformar, imprimir vector.

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:

```
static Vector<Complex>
calculate_fft_generic(Vector<Complex> const &x, bool inverse)
{
  size_t N;
  N = x.size();

  Vector<Complex> X(N);

  // Por defecto se calcula la FFT con estos parámetros:
  double factor = 1;
  int W_phase_sign = -1;

  // En caso de tener que calcular la inversa,
  // modifico el factor de escala y el signo de la fase de W.
  if (inverse)
  {
    factor = 1.0/N;
    W_phase_sign = 1;
  }

  if (N > 1)
  {
    // Divido el problema en 2:
    // Suponemos que la entrada es par y potencia de 2
    Vector<Complex> p(N/2);
    Vector<Complex> q(N/2);
    Vector<Complex> P(N/2);
    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*:

```
template <typename T>
Vector<T>::Vector(int s)
{
  if (s <= 0)
  {
    exit(1);
  }
  else
  {
    size_ = s;
    ptr_ = new T[size_];
  }
}
```

Continuando con la función *calculate_fft_generic()* :

```
    for (size_t i=0; i<N/2; i++)
    {
      p[i] = x[2*i];
```

```
            q[i] = x[2*i+1];
        }

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

        // Combino las soluciones:
        for (size_t k=0; k<N; k++)
        {
            Complex W(cos(k*(2*PI)/N),
                      W_phase_sign*sin(k*(2*PI)/N));
            // Para que se repitan los elementos cíclicamente, se utiliza la función
                módulo
            size_t k2 = k % (N/2);

            X[k] = factor * (P[k2] + W*Q[k2]);
        }
    }
    else
    {
        X = x;
    }

    return X;
}
```

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:

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

$$\boxed{T(N) = 2T(N/2) + N}$$

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:

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

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

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

$$\boxed{T(N) \,\epsilon\, \Theta(N \log N)}$$

Este resultado es coherente, ya que el algoritmo utiliza la técnica de ”divide y vencerász 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 cilco de N iteraciones y 2 sentencias de orden constante, mientras que en el ciclo anidado hay una llamada a una funcion recursiva (*pow_Complex*):

```
static Vector<Complex>
calculate_dft_generic(Vector<Complex> const &x, bool inverse)
{
  Complex aux;
  size_t N;

 N = x.size();

  // Por defecto se calcula la DFT con estos parámetros:
  double factor = 1;
  int W_phase_sign = -1;

  // En caso de tener que calcular la inversa,
  // modifico el factor de escala y el signo de la fase de W.
  if (inverse)
  {
    factor = 1.0/N;
    W_phase_sign = 1;
  }

  Vector<Complex> X(N);

  Complex W(cos((2*PI)/N),
             W_phase_sign*sin((2*PI)/N));

  for(size_t k=0;k<N;k++) {
    for(size_t n=0;n<N;n++) {
      aux += x[n] * pow_complex(W, n*k);
    }
    X[k] = factor * aux;
    aux = 0;
  }
  return X;
}
```

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

```
Complex
pow_complex(Complex const &z, size_t p)
{
  if(!p) return 1;

  if(p == 1){
    return z;
  }
  else{
    Complex aux = pow_complex(z, p/2);
    if(!(p%2))
        return aux * aux;
```

```
        else
          return aux * aux * z;
    }
}
```

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:

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

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

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:

$$\exists\, k \geq 0 \quad / \quad f(p) \;\epsilon\; \Theta(p^{\log_b(a)} \log^k(p))$$

$$\Rightarrow T_p(p) \;\epsilon\; \Theta(p^{\log_b(a)} \log^{k+1}(p))$$

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

$$T_p(p) \;\epsilon\; \Theta(\log p)$$

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:

$$T(N) = (\mathcal{O}(1) + \Theta(\log N)) * N$$

$$\Rightarrow T(N) \;\epsilon\; \Theta(N \log N)$$

Entonces el coste total del primer ciclo es:

$$T(N) = (\mathcal{O}(1) + \Theta(N \log N)) * N$$

$$\Rightarrow T(N) \;\epsilon\; \Theta(N^2 \log N)$$

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

$$T(N) = \mathcal{O}(1) + \Theta(N^2 \log N)$$

$$\Rightarrow \boxed{T(N) \;\epsilon\; \Theta(N^2 \log N)}$$

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:

$$\boxed{T(N) \;\epsilon\; \Theta(N^2)}$$

7. Estructura de archivos

8. Compilación

Como se compila

9. Casos de prueba

los q aparecen en la especificación del tp, mostrar capturas de pantalla de la consola ejecutando todo

10. Código

11. Enunciado