75.04/95.12 Algoritmos y Programación II
Trabajo práctico 1: algoritmos y estructuras de datos

Universidad de Buenos Aires - FIUBA
Primer cuatrimestre de 2021

1. Objetivos

Ejercitar conceptos relacionados con estructuras de datos, diseño y análisis de algoritmos. Escribir un programa en C++ (y su correspondiente documentación) que resuelva el problema que presentaremos más abajo.

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 digital de acuerdo con lo que mencionaremos en la sección 5, y con una copia de los archivos fuente necesarios para compilar el trabajo.

El trabajo y eventuales reentregas deberán ser presentadas exclusivamente a través del campus según las fechas pre-establecidas para el mismo en la planificación del curso presentada en la primera clase del cuatrimestre. Asimismo todas las devoluciones se realizarán en horario de clase, en persona, para lo cual deberán estar presentes todos los integrantes del grupo.

4. Descripción

En este trabajo continuaremos la línea del trabajo práctico anterior: el objetivo perseguido es robustecer y extender nuestras implementaciones de `bignum` introduciendo una serie de mejoras. En particular, en esta oportunidad focalizaremos nuestros esfuerzos en:

- Trabajar con precisión arbitraria (no fija),
- Desarrollar y analizar distintas variantes para multiplicar números grandes, y

- Interpretar y resolver expresiones aritméticas más complejas.

4.1. Algoritmos de multiplicación

Nuestras implementaciones deberán soportar por lo menos dos algoritmos de multiplicación: el algoritmo *standard* desarrollado en el contexto del TP 0, y el algoritmo de *Karatsuba* (abordado en la Sección 4.1.1). Al momento de ejecutar el programa, podrá especificarse cuál algoritmo utilizar a través de una opción de línea de comando (Sección 4.4). A los efectos de lograr implementaciones desacopladas de la representación de los números, utilizaremos el patrón de diseño *Strategy* (Sección 4.1.2) para materializar el concepto de algoritmo de multiplicación.

4.1.1. El algoritmo de Karatsuba

El algoritmo de Karatsuba [3] permite multiplicar números enteros de manera recursiva, expresando a cada operando convenientemente para reducir la cantidad de dígitos en las sucesivas llamadas. En particular, para calcular el producto $a \times b$, el algoritmo se basa en las siguientes descomposiciones, para cierto valor de k:

- $a = a_0\,10^k + a_1$
- $b = b_0\,10^k + b_1$

De este modo, se tiene que

$$\begin{aligned} a \times b &= \left(a_0\,10^k + a_1\right) \times \left(b_0\,10^k + b_1\right) \\ &= a_0 \times b_0\,10^{2k} + \left(a_0 \times b_1 + a_1 \times b_0\right)\,10^k + a_1 \times b_1 \end{aligned}$$

De esta identidad se desprende una estructura recursiva para calcular el producto $\times$ a partir de otros cuatro productos $\times$ que involucran operandos de menor tamaño. No obstante, notando que

$$a_0 \times b_1 + a_1 \times b_0 = (a_0 + a_1) \times (b_0 + b_1) - a_0 \times b_0 - a_1 \times b_1$$

podemos "ahorrar" el cálculo de uno de dichos productos, lo cual en definitiva permite eliminar una llamada recursiva.

Con el objeto de balancear el tamaño de los operandos involucrados en los subproductos (y de esta manera optimizar la complejidad asintótica), el valor de k debe coincidir con $n/2$, siendo n la mínima cantidad de dígitos de a y b.

4.1.2. El patrón de diseño *Strategy*

Un *patrón de diseño* es una solución reutilizable para un problema que ocurre comúnmente en el proceso de diseño de software. Para el caso de *Strategy*, este se trata de un patrón que define una familia de algoritmos, cada uno representado con un objeto, y los hace intercambiables en tiempo de ejecución. *Strategy* permite que el algoritmo varíe en forma independiente del cliente que requiera su utilización [5].

Esto se logra definiendo una interfaz básica y homogénea para los algoritmos involucrados. Siguiendo esta interfaz, los clientes podrán interactuar de igual forma con cada uno de los algoritmos, sin tener que hacer distinciones innecesarias.

En C++, definiremos esta interfaz como una clase *abstracta* (esto es, una clase que no poseerá instancias). Allí se deberá definir el prototipo de cada mensaje que nuestros algoritmos deben saber responder. La implementación concreta de cada uno de ellos se hará en la definición de la clase de cada algoritmo. Por supuesto, cada una de ellas será subclase de la interfaz mencionada.

Los detalles restantes para la correcta implementación de este patrón quedará a criterio de cada grupo.

4.2. Expresiones aritméticas

La lectura e interpretación de las expresiones aritméticas de la entrada la llevaremos a cabo mediante el algoritmo *shunting yard* [2]. Nos limitaremos a decodificar expresiones formadas por números enteros y los operadores aritméticos que deben implementarse (`+`, `-`, `*` y `/`). Además, utilizaremos paréntesis para agrupar subexpresiones cuando sea conveniente. En otras palabras, a los efectos del objetivo del trabajo, no será necesario implementar todas las funcionalidades descriptas en [2] (e.g., llamadas a funciones o uso de variables simbólicas).

4.3. Otras consideraciones

Como mencionamos en la Sección 4, nuestras implementaciones de `bignum` deben operar con **precisión arbitraria**. En otras palabras, los programas deben estar preparados para trabajar con números con una cantidad de dígitos variable. En este sentido, observar que, a diferencia de la problemática abordada en el trabajo práctico anterior, esta vez no tendremos una opción de línea de comando para indicar la precisión requerida.

Por otro lado, nos interesa también implementar la **división entera** entre `bignum`s. Para ello, emplearemos el algoritmo usual de división [4]. Notar que, al ser división entera, nos interesa únicamente buscar el cociente de la división, ignorando el resto de la misma.

Finalmente, todos los algoritmos involucrados en el desarrollo del trabajo deben acompañarse de un análisis de complejidad temporal y espacial de peor caso. Se sugiere complementar este análisis teórico con pruebas empíricas (e.g. mediciones de tiempo de ejecución para operandos con distintas dimensiones), en particular para comparar correctamente las estrategias de multiplicación.

4.4. Línea de comando

Las opciones `-i` y `-o` permiten seleccionar los *streams* de entrada y salida respectivamente. Por defecto, éstos serán `cin` y `cout`. Lo mismo ocurrirá al recibir "`-`" como argumento.

Por otro lado, la opción `-m` indicará el algoritmo de multiplicación que se desea utilizar. Las dos opciones posibles son `standard` y `karatsuba`, siendo este último el valor por defecto en caso de que la opción no esté presente.

Al finalizar, todos nuestros programas retornarán un valor nulo en caso de no detectar ningún problema; en caso contrario, devolveremos un valor no nulo (por ejemplo `1`).

4.5. Formato de los archivos de entrada y salida

El formato a adoptar para el *stream* de entrada consiste en una secuencia de cero o más expresiones aritméticas, cada una ocupando una línea distinta. A su vez, la separación entre los operandos y los operadores puede darse con cero o más espacios, entendiendo por tales a los caracteres `SP`, `\f`, `\r`, `\t`, `\v`.

Por otro lado, el *stream* de salida deberá listar en líneas distintas los números resultantes de evaluar cada expresión, manteniendo el mismo orden de las operaciones de la entrada.

A continuación, presentaremos algunos ejemplos puntuales de estos archivos.

4.6. Ejemplos

Primero, usamos un archivo vacío como entrada del programa:

```
$ ./tp1 -i /dev/null
```

Notar que la salida es también vacía.

En el siguiente ejemplo, pasamos una única operación como entrada. La salida deberá contener sólo una línea con el resultado adecuado:

```
$ echo 1123581321345589*123456789 | ./tp1
138713742113703577253721
```

No olvidemos que los números son *enteros*: también pueden ser negativos. El siguiente ejemplo muestra un correcto comportamiento del programa frente a estos casos:

```
$ echo "-1  - 5" | ./tp1
-6
```

El siguiente ejemplo emplea varias expresiones aritméticas de mayor complejidad:

```
$ cat input.txt
128222227961264 * (11374617 - 1897262 + 1874)
217836123668736 / 72366 / 376 / 2
(127836312 / 2186716 * 266) * (18613 / 2 * 7625215)
$ ./tp1 -i input.txt -o output.txt
$ cat output.txt
1215447861735024585456
4002925
1094774749188120
```

Observar que el resultado es equivalente si cambiamos el algoritmo de multiplicación subyacente:

```
$ ./tp1 -i input.txt -o output.txt -m standard
$ cat output.txt
1215447861735024585456
4002925
1094774749188120
```

4.7. Portabilidad

A los efectos de este primer trabajo, es deseable (pero no requisito) que la implementación desarrollada provea un grado mínimo de portabilidad. Sugerimos verificar nuestros programas en Windows/MS-DOS y/o alguna versión reciente de UNIX: BSD, o Linux[1].

5. Informe

El informe deberá incluir, como mínimo:

- Una carátula que incluya los nombres de los integrantes y el listado de todas las entregas realizadas hasta ese momento, con sus respectivas fechas.
- Documentación relevante al diseño e implementación del programa.
- Documentación relevante a los algoritmos y estructuras de datos involucrados en la solución del trabajo.
- El análisis de las complejidades solicitado en la Sección 4.3.
- 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.

6. Fechas

La última fecha de entrega y presentación será el **jueves 17 de junio**.

Referencias

[1] Wikipedia, "Long multiplication." `https://en.wikipedia.org/wiki/Multiplication_algorithm#Long_multiplication`.

[1] RedHat, SuSe, Debian, Ubuntu.

[2] Wikipedia, "Karatsuba algorithm." https://en.wikipedia.org/wiki/Karatsuba_algorithm.

[3] Wikipedia, "Shunting yard algorithm." http://en.wikipedia.org/wiki/Shunting_yard_algorithm.

[4] Wikipedia, "Long division." https://en.wikipedia.org/wiki/Long_division.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.