Análisis y Diseño de Algoritmos I

Práctica 3: Medida del tiempo de ejecución

27 de noviembre de 2006

Índice

1. Introducción

Para resolver un problema, pueden existir, en general, distintos algoritmos disponibles. La cuestión está en compararlos y elegir aquél que resulte más adecuado.

En la práctica, esta elección no es sencilla y suele estar condicionada por diversos factores, que perfilan el significado de qué es lo que se entiende por «más adecuado» en una determinada situación. Normalmente, los factores más apremiantes serán aquéllos relacionados con el consumo de algún tipo de recurso: en la mayor parte de las ocasiones, memoria y, sobre todo, *tiempo de ejecución*.

Es en este último recurso en el que centraremos nuestra atención. El enfoque empírico de análisis de algoritmos es, en esencia, muy simple: consiste en programar los distintos algoritmos y, como si de una competición se tratara, probar cada uno de ellos sobre diferentes ejemplares del problema a resolver para comparar sus tiempos de ejecución. Para ello es necesario aprender a realizar correctamente la medida de dichos tiempos.

Tomaremos como ejemplo dos algoritmos sencillos para calcular números de Fibonacci y conduciremos una serie de experimentos a partir de los que será posible establecer conclusiones sobre la eficiencia temporal de ambos.

Posteriormente, justificaremos estos resultados a través de la teoría disponible. Esto es absolutamente necesario, ya que los resultados empíricos por sí mismos no bastan.

2. Tipos de medida

Una magnitud física, como, por ejemplo, el tiempo de ejecución de un programa, es el producto de un valor numérico por una unidad o magnitud de referencia.

El objeto de la *medida* es la determinación del valor numérico de la magnitud física en cuestión. Para ello, será necesario emplear un proceso de medida que permita obtener dicho valor. Pero hay que tener en cuenta que ningún proceso de medida es absolutamente exacto y que, por tanto, el valor real de una magnitud es imposible de determinar. Todo lo más que podemos esperar es obtener medidas lo suficientemente aproximadas para nuestros propósitos.

Una *medida directa* es aquélla en la que el valor de la magnitud a medir se obtiene directamente. Por lo tanto, en una medida de este tipo se observa directamente el valor numérico en un instrumento de medida adecuado.

Una *medida indirecta* es aquélla en la que el valor de la magnitud a medir se obtiene a partir de las medidas de otras magnitudes relacionadas con ella. En consecuencia, una medida de este tipo implica la manipulación matemática de una o varias medidas directas.

3. Factores que influyen en la medida

La diferencia entre el valor real y el valor medido es el *error* de la medida o, más precisamente, su *error absoluto*. El valor exacto del error es siempre desconocido, ya que el valor real de

la magnitud medida también lo es, pero su valor absoluto puede acotarse: esta cota es la *incertidumbre* de la medida. Abusando del lenguaje, se suele emplear también el término «error» para indicar la incertidumbre.

Al error de una medida pueden contribuir diversas causas que, según su naturaleza, podemos clasificar en dos categorías: sistemáticas y accidentales.

Los *errores sistemáticos* tienen su origen en defectos del método o proceso de medida, o del instrumento utilizado que dan lugar a una desviación de los resultados que se produce permanentemente y en un mismo sentido.

Los *errores accidentales* tienen su origen en causas ajenas a nuestro control que alteran los resultados a veces por defecto y otras por exceso. Habitualmente, ya que son impredecibles, se les supone aleatoriamente distribuidos y, en tal caso, se les denomina *errores aleatorios*.

La *resolución* de un instrumento de medida es el valor mínimo que ha de tener la magnitud a medir para poder ser detectada por el instrumento. Todo instrumento tiene una resolución por debajo de la cual el error cometido puede ser tan grande como el valor medido haciendo que éste sea completamente inútil.

También hay que tener en cuenta que el acto de medir influye siempre en la magnitud medida; esto introduce cierta incertidumbre en nuestros resultados. Cuanto mayores sean las magnitudes a medir, menor será en comparación la incertidumbre intrínseca a la realización de la medida.

4. Formas de medir el tiempo

Al analizar empíricamente un algoritmo es necesario programarlo y realizar una serie de experimentos con distintos ejemplares de entrada, midiendo los tiempos de ejecución resultantes.

Para medir el tiempo de ejecución de un programa completo puede emplearse la orden `time`, presente en los sistemas operativos tipo Unix, que permite obtener, generalmente, información adicional sobre otros recursos consumidos por la ejecución del programa. Sin embargo, este método es muy poco flexible: en muchas ocasiones estamos interesados en medir el tiempo de un determinado fragmento de código que no deseamos aislar en un programa independiente.

Cabe también la posibilidad de emplear un *perfilador*. No obstante, los perfiladores pueden ser complicados de manejar, y sus datos más difíciles de interpretar y de postprocesar (para obtener gráficas, etc.). Por último, no es sencillo encontrar un buen perfilador para C++.

Otra posibilidad, que es la que emplearemos, consiste en medir el tiempo de ejecución «por diferencia» empleando la función *clock*() de la biblioteca estándar de C++, declarada en <*ctime*>, al igual que el tipo *clock_t* y la macro *CLOCKS_PER_SEC*. Esta función, heredada de C, devuelve la mejor aproximación disponible del tiempo del procesador empleado por el programa desde que éste comenzó o, al menos, desde un cierto tiempo relacionado exclusivamente con la ejecución del programa. El resultado es un *clock_t*, un tipo aritmético entero que depende del sistema, cuyas unidades se denominan *clocks* o «pulsos». Para hallar el tiempo en segundos se divide su resultado entre el número de pulsos por segundo, que está disponible a través de *CLOCKS_PER_SEC*.

Como, normalmente, lo que deseamos calcular es el tiempo de procesador transcurrido entre dos puntos del programa, hay que obtener primero la diferencia entre el número de pulsos devueltos por *clock*() en los instantes inicial y final. Para simplificar todo el proceso, podemos utilizar la clase *cronometro* que se presentó en la primera práctica. Esta clase nos permite crear un cronómetro que podemos activar y parar a voluntad. En cualquier momento podemos comprobar el tiempo transcurrido desde su última activación o, si lo hemos parado, el tiempo medido desde su última activación hasta el momento en que se paró.

Así, para realizar una medida directa del tiempo que tarda un determinado fragmento de código utilizando un objeto cronómetro, basta hacer:

```
cronometro c;
c.activar();
// ... fragmento a cronometrar ...
c.parar();
double t = c.tiempo();
```

No obstante, esto no es adecuado si el tiempo a medir no es lo suficientemente grande comparado con la resolución del cronómetro. Hay que evitar realizar medidas directas en tales situaciones. En nuestro sistema, la resolución que arroja *clock*() es de diez milésimas de segundo, aproximadamente.[1] Ciertamente, no es mucho. En arquitecturas especialmente diseñadas para que la medida del tiempo sea precisa se consiguen resoluciones del orden del microsegundo.

Para solventar el problema se puede utilizar la siguiente técnica de medida indirecta: medir el tiempo total de un cierto número de repeticiones del experimento y dividirlo entre el número de repeticiones realizadas para obtener el tiempo del experimento. De este modo, si el número de repeticiones es lo suficientemente alto, el tiempo medido directamente será notablemente superior a la resolución del instrumento de medida y se disminuirá el error.

Es muy importante que el tiempo de cada repetición sea el mismo porque, de lo contrario, el resultado final puede no ser significativo. Si el código es determinista, esto implica emplear exactamente los mismos datos de entrada en cada repetición.

Por ejemplo, sabiendo que la resolución es 0,01 s, si queremos obtener el tiempo de un experimento con una precisión de 1 ms hemos de medir el tiempo total de 10 repeticiones de ese mismo experimento y dividirlo entre 10. Del mismo modo, necesitaremos 10 000 repeticiones para conseguir una precisión de 1 μs.

```
const int r = 10000;
cronometro c;
c.activar();
for (int i = 0; i < r; ++i) {
  // ... fragmento a cronometrar ...
}
c.parar();
double t = c.tiempo() / r;
```

[1] En Linux, la tarea de reloj programa el temporizador para generar interrupciones con una frecuencia de 100 Hz, es decir, con un periodo de $1/100 = 0{,}01$ s.

Sin embargo, repetir un experimento un número de veces elevado puede hacer que el tiempo total de experimentación sea muy alto. El problema radica en que no se suele tener, a priori, una idea del tiempo que va a tardar el experimento y, en consecuencia, tampoco se conoce el número de repeticiones adecuado.

En vez de repetir cada experimento un número fijo de veces, se puede emplear un *esquema adaptativo de medida*, repitiendo el experimento durante el tiempo necesario para garantizar la realización de un número suficiente de repeticiones y promediando el tiempo total entre dicho número.

Por ejemplo, si el máximo error absoluto que se comete es 0,01 (supongamos que el único error cometido se debe a la resolución del cronómetro) y se desea garantizar un error relativo máximo del 1‰ el tiempo total mínimo de experimentación sería 0,01/0,001 + 0,01:

```
cronometro c;
long int r = 0;
const double e_abs = 0.01, // Máximo error absoluto cometido.
             e_rel = 0.001; // Máximo error relativo aceptado.
c.activar();
do {
  // ... fragmento a cronometrar ...
  ++r;
} while (c.tiempo() < e_abs / e_rel + e_abs);
c.parar();
double t = c.tiempo() / r;
```

La realización de repeticiones tiene otro efecto beneficioso, ya que ayuda también a disminuir cierto tipo de errores accidentales, como los debidos a la aparición puntual de un pico en la carga del sistema. Téngase en cuenta que en un sistema multitarea, una situación de carga del sistema excesiva puede hacer variar los resultados ofrecidos por *clock*().

Al analizar el tiempo para distintos ejemplares de entrada, puede ocurrir que los ejemplares de mayor tamaño tengan tiempos apreciables de por sí y sea un desperdicio repetir su ejecución, mientras que los de menor tamaño tengan tiempos despreciables y requieran muchas repeticiones. En tal caso, utilice un esquema adaptativo.

5. Elección de los ejemplares de prueba

El tiempo que tarda un algoritmo en ejecutarse puede depender de la entrada concreta que reciba y no sólo del tamaño de ésta. El hecho de que muchos algoritmos se analicen respecto al tamaño de la entrada responde a una simplificación práctica necesaria, pero no olvide que para ejemplares de entrada del mismo tamaño los tiempos pueden ser muy distintos.

Es por ello, que se distinguen diversos casos de análisis: mejor caso, peor caso y caso promedio. Éstos permiten clasificar las posibles entradas en familias que poseen para cada tamaño tiempos de ejecución similares.

Por lo tanto, antes de analizar empíricamente un algoritmo en función del tamaño de su

entrada hay que tener claro qué caso se desea analizar y cuáles son los ejemplares de entrada que lo producen; entonces podrán realizarse los experimentos a partir de una muestra de dichos ejemplares. Al realizar un análisis en el caso promedio los ejemplares de prueba se seleccionan aleatoriamente, pero es importante tener en cuenta que su distribución de probabilidad puede influir en los resultados.

Insistimos en que si se realiza una medida indirecta del tiempo de un algoritmo tenga siempre la precaución de repetir su ejecución **para el mismo ejemplar de prueba**. Por ejemplo, si el algoritmo modifica su entrada puede ser necesario almacenar ésta para poder realizar la siguiente repetición en igualdad de condiciones.

Igualmente, cuando se van a comparar dos algoritmos distintos que resuelven un mismo problema hay que ejecutarlos exactamente sobre los mismos ejemplares de prueba, no basta hacerlo sobre ejemplares del mismo tamaño.

Las indicaciones anteriores son muy importantes. Sólo siguiéndolas resultará posible comparar los resultados experimentales obtenidos en el laboratorio con los teóricos.

6. Ejemplo: números de Fibonacci

Al objeto de ilustrar lo explicado en secciones anteriores nos disponemos a analizar empíricamente la eficiencia temporal de dos algoritmos para calcular números de Fibonacci.

Escribiremos una serie de programas y emplearemos diversas herramientas que nos ayudarán a medir los tiempos de los algoritmos para distintos valores de la entrada y a trazar gráficas que nos faciliten su comparación. Finalmente, explicaremos los datos experimentales obtenidos en función de los resultados teóricos correspondientes.

Recuérdese la definición de F_n, el n-ésimo número de Fibonacci:

$$F_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F_{n-1} + F_{n-2}, & n > 1. \end{cases}$$

El primer algoritmo se obtiene trivialmente a partir de esta definición recursiva. El segundo, algo más elaborado, resulta de aplicar *programación dinámica* a dicha definición, una técnica que estudiaremos en la asignatura de análisis y diseño de algoritmos del segundo cuatrimestre.

```
f : n → r
si n ≤ 1
   r ← n
si no
   r ← f(n − 1) + f(n − 2)
```

```
f : n → r
⟨a, b⟩ ← ⟨0, 1⟩
mientras n > 1
          c ← a + b
          ⟨a, b⟩ ← ⟨b, c⟩
          n ← n − 1
si n = 0
   r ← a
si no
   r ← b
```

6.1. Organización del código

El trabajo se organiza en una serie de ficheros que se distribuyen junto a la práctica:

makefile *Makefile* que incorpora, entre otros, los siguientes objetivos:

all — Es el objetivo por omisión, es decir, el que se activa cuando se ejecuta `make` sin más. Se encarga de hacer todo lo necesario para obtener los ficheros de datos de las gráficas.

graficas — Muestra las gráficas.

graficas-eps — Crea los ficheros EPS correspondientes a las gráficas.[2]

clean — Limpia el directorio de trabajo eliminando todo lo que puede volver a ser generado, salvo los ficheros de datos.

clean-all — Limpia completamente el directorio de trabajo.

graficas.plot Guión para GNU Plot que se emplea para mostrar las gráficas.

graficas-eps.plot Guión para GNU Plot que se emplea para crear los ficheros EPS.

natural.h Contiene el tipo empleado para representar números naturales.

fibonacci-1.cpp Contiene la función correspondiente al primer algoritmo.

prueba-1.cpp Programa que realiza los experimentos correspondientes al primer algoritmo y escribe los resultados en la salida estándar.

fibonacci-2.cpp Contiene la función correspondiente al segundo algoritmo.

prueba-2.cpp Programa que realiza los experimentos correspondientes al segundo algoritmo y escribe los resultados en la salida estándar.

En las siguientes secciones se discutirá el contenido de cada uno de estos ficheros en detalle.

6.2. Makefile

La utilidad GNU Make está especialmente diseñada para ayudar a rehacer trabajos monótonos que deban repetirse cada vez que ciertos ficheros sean modificados. Ésta es, precisamente, la situación que nos encontraremos en muchas ocasiones.

Para ello, se describen en un fichero una serie de *reglas*. Cada regla consta de *objetivos*, *dependencias* y *órdenes*. Sólo los objetivos son obligatorios en una regla, y se separan de las dependencias por el carácter «:». Cada línea que describe las dependencias de uno o varios objetivos puede ir seguida de una serie de órdenes sangradas obligatoriamente con un tabulador.

Cuando se requiera la realización de un objetivo, basta ejecutar `make`, que comprobará si alguna de sus dependencias ha sido modificada con posterioridad al objetivo (esto es, si el

[2] EPS significa «PostScript encapsulado» (*Encapsulated PostScript*). PostScript© es un lenguaje creado por Adobe® y prácticamente universal. Muchas impresoras modernas aceptan directamente este lenguaje.

objetivo es más antiguo). En tal caso, el objetivo deberá rehacerse y sus órdenes ejecutarse, no sin antes comprobar y rehacer las dependencias del mismo que sean necesarias.

No todos los objetivos son necesariamente ficheros: existen «objetivos falsos», que no están asociados a ningún fichero, y que se emplean para desencadenar una serie de acciones (por ejemplo, la limpieza del directorio de trabajo). Los objetivos falsos *all*, *clean* y *clean-all* son tan comunes en los *makefiles* que no se suelen traducir sus nombres.

Listados/Fibonacci/makefile

```
# Compilador de C++ y opciones de compilación.

CXX = c++
CXXFLAGS = -ansi -Wall -O$(OPTIMIZACION)

# Nivel de optimización (por omisión, no se optimiza).

OPTIMIZACION = 0

# Módulos objeto y ejecutables.

OBJS = prueba-1.o prueba-2.o fibonacci-1.o fibonacci-2.o
EXES = prueba-1 prueba-2

# Ficheros de tiempo y de gráficas.

TIEMPOS = prueba-1.tmp prueba-2.tmp
GRAFICAS = prueba-1.eps prueba-2.eps pruebas-1-y-2.eps

# Por omisión, obtiene los ficheros de tiempo.

all: $(TIEMPOS)

# Obtención de los ejecutables.

prueba-1: prueba-1.o fibonacci-1.o
	$(CXX) $(LDFLAGS) -o $@ $^

prueba-2: prueba-2.o fibonacci-2.o
	$(CXX) $(LDFLAGS) -o $@ $^

# Obtención de los objetos.

$(OBJS): natural.h

prueba-1.o prueba-2.o: ../cronometro/cronometro.h

# Obtención de los ficheros de tiempo.
```

```
39
40 prueba-1.tmp: prueba-1
41         ./$< | tee $@
42
43 prueba-2.tmp: prueba-2
44         ./$< | tee $@
45
46 # Obtención de las gráficas.
47
48 graficas:
49         gnuplot graficas.plot
50
51 graficas-eps:
52         gnuplot graficas-eps.plot
53
54 # Limpieza del directorio.
55
56 clean:
57         $(RM) $(EXES) $(OBJS) *~
58
59 clean-all: clean
60         $(RM) $(TIEMPOS) $(GRAFICAS)
```

Al principio, se definen una serie de variables a cuyos valores se accede mediante *$*. Algunas de ellas, como *CXX* y *CXXFLAGS*, son especiales: `make` las utiliza en sus «reglas implícitas». Por ejemplo, una de las reglas implícitas de `make` dice que un módulo objeto se obtiene a partir de su fuente *.cpp* compilando con *$(CXX)* y con las opciones *$(CXXFLAGS)*. Por lo tanto, `make` conoce esta información y no es necesario suministrársela.

Existen otras variables especiales. Así, **$@** es el objetivo que se está tratando, *$^*, todas sus dependencias, y *$<* su primera dependencia.

Los valores de las variables se pueden cambiar desde fuera. Por ejemplo, para compilar de manera que luego se pueda emplear el depurador se haría:

```
% make CXXFLAGS=-g
```

Se ha incluido una variable llamada *OPTIMIZACION*, que inicialmente vale 0, para poder cambiar fácilmente el nivel de optimización del compilador. Para GNU C++, 0 indica «sin optimización», y 3, «optimización plena». Por ejemplo, si ya se ha compilado el código para llevar a cabo un experimento sin optimización y, posteriormente, se desea recompilarlo para volver a realizar el experimento con optimización plena, puede hacerse:

```
% make clean
% make OPTIMIZACION=3
```

6.3. Gráficas

Emplearemos la herramienta GNU Plot para representar gráficamente los resultados obtenidos del análisis empírico de los algoritmos. Esta herramienta es muy completa, y merece la pena estudiarla por sí misma; para más información, escriba la orden **help** tras ejecutar `gnuplot`; también puede consultar la información disponible en el sistema GNU Info.

El siguiente fichero recoge una sesión de GNU Plot cuyo objetivo es mostrar las gráficas correspondientes a los datos almacenados en los ficheros *.tmp*. Las órdenes que contiene pueden ser introducidas desde dentro del programa; no obstante, resulta muy cómodo y útil agruparlas en un *guión* y pasar éste como parámetro desde el exterior al ejecutar `gnuplot`. Esto es lo que se hace exactamente dentro del *makefile*.

Listados/Fibonacci/graficas.plot

```
1  # Título de cada eje.
2
3  set xlabel "n"
4  set ylabel "t(n) (tiempo en segundos)"
5
6  # Estilo de presentación (puntos interpolados linealmente).
7
8  set data style linespoints
9
10 # Representación gráfica.
11
12 set terminal x11 1
13 plot "prueba-1.tmp" title "Fibonacci recursivo trivial"
14
15 set terminal x11 2
16 plot "prueba-2.tmp" title "Fibonacci dinámico"
17
18 set terminal x11 3
19 plot "prueba-1.tmp" title "Fibonacci recursivo trivial", \
20      "prueba-2.tmp" title "Fibonacci dinámico"
21
22 # Pausa (hasta que se pulse [Intro]).
23
24 pause -1 "[Intro] para terminar\n"
```

El formato de los ficheros de datos es muy sencillo. Son ficheros de texto donde cada línea contiene la abscisa y la ordenada de un punto, como en el siguiente fragmento de ejemplo:

```
...     ...
41      10.22
42      16.53
43      26.75
44      43.28
45      70.02
```

La opción **terminal** controla el dispositivo gráfico a través del que se generarán las gráficas. El dispositivo especial *x11* representa los datos en una ventana gráfica. Como se observa, es posible crear varias ventanas gráficas numerándolas. También se puede representar más de una gráfica simultáneamente en cada ventana, lo que resulta muy útil a efectos de comparación.

Dado que puede ser interesante obtener gráficas en formatos apropiados para su impresión e inclusión en otros documentos (como, éste), resulta útil preparar un guión específico.

Listados/Fibonacci/graficas–eps.plot

```
1  # Codificación ISO Latin−1 y terminal EPS.
2
3  set encoding iso_8859_1
4  set terminal postscript eps solid
5
6  # Título de cada eje.
7
8  set xlabel "n"
9  set ylabel "t(n) (tiempo en segundos)"
10
11 # Estilo de presentación (puntos interpolados linealmente).
12
13 set data style linespoints
14
15 # Creación de los ficheros EPS.
16
17 set output "prueba-1.eps"
18 plot "prueba-1.tmp" title "Fibonacci recursivo trivial"
19
20 set output "prueba-2.eps"
21 plot "prueba-2.tmp" title "Fibonacci dinámico"
22
23 set output "pruebas-1-y-2.eps"
24 plot "prueba-1.tmp" title "Fibonacci recursivo trivial", \
25      "prueba-2.tmp" title "Fibonacci dinámico"
```

Este guión es muy parecido al anterior. Las líneas clave son las que permiten cambiar la codificación de los caracteres, el dispositivo a través del que se generan los gráficos y el destino de la propia salida a la que se envían éstos.

Gracias a la opción *encoding* es posible obtener correctamente las representaciones gráficas correspondientes a los caracteres de códigos distintos del ASCII (ISO 646). Empleamos el código ISO Latin-1 (ISO 8859-1), oficial en los países de Europa Occidental.

Obtendremos código PostScript y, más concretamente, EPS. Esto se consigue cambiando la opción **terminal** que en nuestro sistema, por omisión, es *x11*. Por último, con la opción **output** se redirige la salida estándar al fichero indicado. Por omisión, los datos se envían a la salida estándar (salvo en el caso de dispositivos especiales como *x11*).

6.4. Programas

La programación de los algoritmos es directa y no merece mayor explicación, salvo por la elección del tipo *natural*.

Listados/Fibonacci/fibonacci–1.cpp

```
#include "natural.h"

// Cálcula el n−ésimo número de Fibonacci por el algoritmo recursivo trivial.

natural f(natural n)
{
  return n <= 1 ? n : f(n − 1) + f(n − 2);
}
```

Listados/Fibonacci/fibonacci–2.cpp

```
#include "natural.h"

// Cálcula el n−ésimo número de Fibonacci por programación dinámica.

natural f(natural n)
{
  natural a = 0, b = 1;
  while (n > 1) {
    const natural c = a + b;
    a = b;
    b = c;
    −−n;
  }
  return n ? b : a;
}
```

En primera instancia, consideraremos que los números naturales tienen precisión limitada. Así, una posibilidad es que *natural.h* contenga:

typedef unsigned long int *natural*;

Emplear precisión limitada es ciertamente una limitación, y valga la redundancia, puesto que no nos permite calcular exactamente más que unos cuantos números de Fibonacci (recuérdese que el número de dígitos de F_n crece linealmente con n).

Visto de otro modo: ya que cada número se almacena en un **unsigned long int**, esto equivale a realizar los cálculos modularmente; no se calcula F_n, sino $F_n \mod p$, siendo p el siguiente número natural al máximo valor representable con este tipo.

Si no es esto lo que queremos y se desean números naturales de precisión ilimitada, es necesario programar una clase para su manejo o emplear una biblioteca especializada, ya que C++ no la posee primitivamente.

6.5. Pruebas realizadas

Analizaremos el tiempo de ejecución en función de n, el valor de la entrada. Decidimos variar n desde 0 hasta 45 para cada algoritmo. Como se podrá comprobar, el tiempo total del experimento es apreciable.

En el caso del primer algoritmo podemos emplear una medida directa, aunque ésta sea completamente imprecisa para los primeros valores de n, ya que rápidamente el tiempo de cálculo comienza a ser apreciable.

Listados/Fibonacci/prueba–1.cpp

```
1  #include <iostream>
2  #include "natural.h"
3  #include "../cronometro/cronometro.h"
4
5  natural f(natural n);
6
7  // Genera la tabla de tiempos frente a n.
8
9  int main()
10 {
11   cronometro c;
12   for (natural n = 0; n <= 45; ++n) {
13     c.activar();
14     f(n);
15     c.parar();
16     std::cout << n << '\t' << c.tiempo() << std::endl;
17   }
18 }
```

En el caso del segundo algoritmo los tiempos de cálculo son tan pequeños que la medida no debe realizarse directamente. Esto se debe a la escasa resolución de nuestro instrumento de medida. Podemos emplear la técnica de medida indirecta explicada repitiendo cada experimento 10 000 000 de veces para obtener una precisión de 1 ns.

Listados/Fibonacci/prueba–2.cpp

```
1  #include <iostream>
2  #include "natural.h"
3  #include "../cronometro/cronometro.h"
4
5  natural f(natural n);
6
7  // Genera la tabla de tiempos frente a n.
8
9  int main()
10 {
11   cronometro c;
```

```
12    for (natural n = 0; n <= 45; ++n) {
13      const long int r = 10000000;
14      c.activar();
15      for (long int i = 0; i < r; ++i)
16        f(n);
17      c.parar();
18      std::cout << n << '\t' << c.tiempo() / r << std::endl;
19    }
20  }
```

6.6. Resultados obtenidos

Los experimentos se han realizado con el equipo consignado en la siguiente tabla. Por supuesto, al repetir estos experimentos en el laboratorio o en su casa, presumiblemente con un equipo distinto y bajo condiciones también distintas, obtendrá datos diferentes.

Microprocesador	Pentium III 933 MHz
Memoria principal	SDRAM 133 MHz, 512 MiB
Sistema operativo	SuSE Linux 9.0 (2.4.21)
Compilador	GNU C++ 3.3.1

Los resultados obtenidos para cada experimento aparecen en las gráficas que se presentan en la figura 1. Basta echar un vistazo a $1c$ para comprobar cómo el tiempo empleado por el segundo programa es cada vez más despreciable frente al correspondiente al primero.

Se observa un comportamiento exponencial de la función representada en $1a$. Esto coincide plenamente con los resultados teóricos, ya que no es difícil comprobar que se realizan $\Theta(\phi^n)$ sumas y que ninguna otra operación se ejecuta asintóticamente más que la suma. Cuando cada suma se ejecuta empleando aritmética de precisión limitada (y así lo hace el programa), es una operación elemental y el tiempo real del programa asociado coincide, en orden, con el abstracto del algoritmo.

En efecto, sea $t(n)$ el número de sumas que realiza el primer algoritmo, su ecuación de recurrencia asociada es:

$$t(n) = \begin{cases} 0, & n \leq 1 \\ t(n-1) + t(n-2) + 1, & n > 1. \end{cases}$$

La solución viene descrita por

$$t(n) = \frac{1}{\phi + \phi^{-1}} \cdot (\phi^{n+1} - \phi^{-(n+1)}) - 1,$$

siendo ϕ el *número áureo*. Por lo tanto, $t(n) \in \Theta(\phi^n)$.

Por el contrario, la función representada en $1b$ tiene un comportamiento lineal. Esto se debe a que $t(n) = n - 1$ sumas (para $n \geq 1$) en el caso del segundo algoritmo. Por idénticas razones a las ya comentadas, el tiempo real del programa asociado pertenecerá a $\Theta(n)$.

Ejercicios

1. Como se ha explicado, los programas presentados calculan los números de Fibonacci con precisión limitada. ¿Cuál es el mayor número de Fibonacci que puede calcular exactamente en su sistema? Compruebe qué valor se obtiene en lugar del número de Fibonacci que lo sigue y explique exactamente por qué se produce ese valor y no otro.

2. Los tiempos en segundos obtenidos en el equipo reseñado para los últimos valores de n y cada algoritmo aparecen en la siguiente tabla:

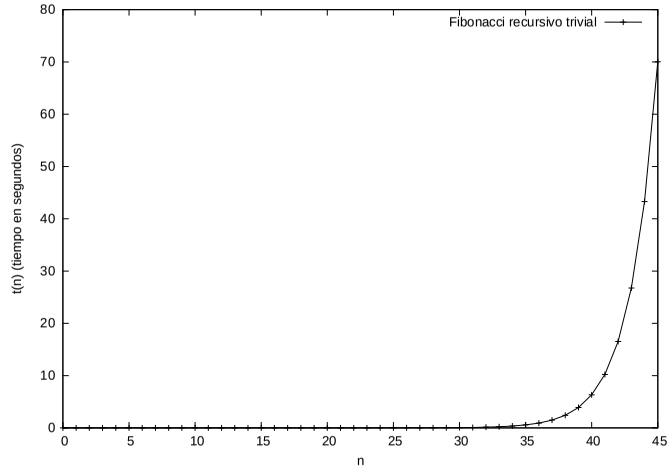
n	t_1	t_2
41	10,22	$5{,}49 \cdot 10^{-7}$
42	16,53	$5{,}64 \cdot 10^{-7}$
43	26,75	$5{,}75 \cdot 10^{-7}$
44	43,28	$5{,}87 \cdot 10^{-7}$
45	70,02	$6{,}00 \cdot 10^{-7}$

Repita los experimentos en el laboratorio. Debe observar que los nuevos tiempos obtenidos se diferencian de los anteriores, aproximadamente, en una constante multiplicativa. ¿Por qué? Calcule dicha constante y el porcentaje de mejora o empeoramiento que se produce en los tiempos de cada algoritmo.

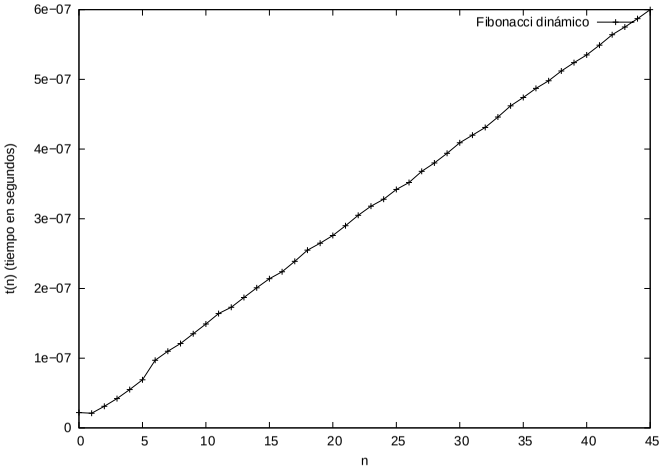
3. Repita los experimentos con *OPTIMIZACION*=3 y compruebe el impacto que la activación de la opción de optimización plena del compilador tiene sobre los tiempos de ejecución de cada algoritmo.

4. Repita los experimentos empleando un esquema adaptativo de medida y compruebe gráficamente que obtiene resultados similares.

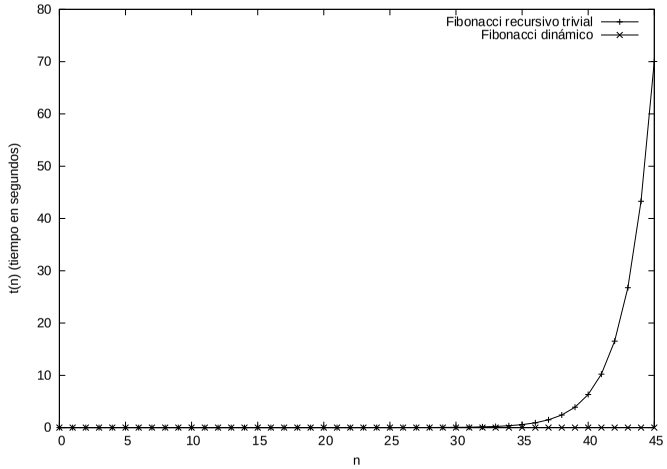
Figura 1: Resultados de los experimentos.



a) Tiempos obtenidos con el primer algoritmo.



b) Tiempos obtenidos con el segundo algoritmo.



c) Tiempos obtenidos con ambos algoritmos.