

Práctica 3: Medición experimental de tiempos de ejecución en programas

Objetivos:

- Conocer en la práctica la diferencia en tiempo de ejecución entre implementaciones de distinta complejidad temporal de distintas operaciones sobre un vector dinámico.

Programa a realizar:

Debéis desarrollar un programa que genere iterativamente vectores dinámicos de números (tipo int) de distintos tamaños, y realice las siguientes operaciones sobre los mismos:

- Impresión por pantalla de todos los pares de números que sumen una determinada cantidad (por ejemplo, 100).
- Ordenación rápida (*quicksort*) del vector dinámico
- Ordenación por el método de inserción del vector dinámico

El programa deberá medir el tiempo necesario para realizar cada operación y guardar los resultados en un fichero. Usaremos dicho fichero para realizar un análisis básico de los resultados obtenidos, mediante su representación gráfica.

Medición de tiempos de ejecución:

Para medir el tiempo de ejecución de un programa disponemos en C de la biblioteca `<time.h>`. Dicha biblioteca dispone de funciones/procedimientos, constantes, tipos de datos y registros que permiten evaluar el número de períodos o “tics” de reloj que consume una secuencia de instrucciones. La función **clock()** permite acceder a dicha información. Su prototipo es el siguiente:

clock_t clock (void)

clock() devuelve el número de “tics” de reloj que han transcurrido desde un determinado evento de referencia (la creación del proceso desde donde se invoca). Mediante **clock()** podemos medir el tiempo transcurrido en una secuencia de instrucciones sin más que invocar dicha función al inicio y al final de dicha secuencia, guardar ambos resultados y restarlos.

El tipo de dato que devuelve **clock()** es el tipo opaco **clock_t**, que en nuestro caso equivale a **long int** (aunque en otros sistemas puede ser **double**).

Para convertir el número de periodos de reloj a segundos de forma fácil y general puede utilizarse la constante

int CLOCKS_PER_SEC;

Práctica 3: Medición experimental de tiempos de ejecución en programas

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 int main(int argc, char** argv) {
7     clock_t inicio=-1, fin=-1;
8
9     inicio=clock(); //TOMAMOS LA PRIMERA REFERENCIA
10    /* AQUÍ SE INICIA EL BLOQUE DE INSTRUCCIONES CUYO TIEMPO DE EJECUCIÓN QUEREMOS MEDIR */
11    AQUÍ VA EL CÓDIGO QUE QUEREMOS
12    "CRONOMETRAR"
13
14    /* AQUÍ FINALIZA EL BLOQUE DE INSTRUCCIONES CUYO TIEMPO DE EJECUCIÓN QUEREMOS MEDIR */
15    fin=clock(); //TOMAMOS LA SEGUNDA REFERENCIA
16
17    //IMPRIMIMOS EL RESULTADO
18    printf("%u\t%lf\n", fin-inicio, (fin-inicio)/(double)CLOCKS_PER_SEC);
19
20    return (EXIT_SUCCESS);
21 }

```

Ejemplo: medida experimental de tiempos en la implementación recursiva de la sucesión de Fibonacci.

En este ejemplo debéis ejecutar el programa fibonacci.c, que incluye la implementación recursiva de la serie de Fibonacci. Dicha serie se define como:

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

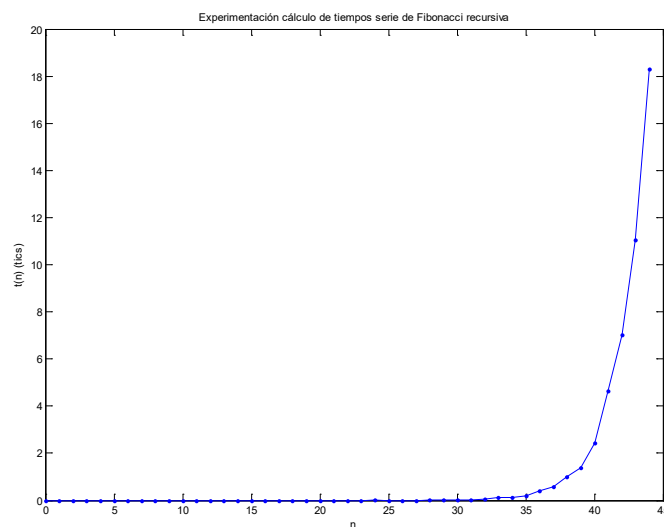
Podéis verificar experimentalmente como progresivamente la obtención de los resultados se ralentiza (debido a que el número de llamadas recursivas, y por lo tanto el tiempo de ejecución, crece exponencialmente)¹. Fijaos, en particular en el tiempo de obtención de los últimos términos de la sucesión (n=43, 44).

¹ Como curiosidad, comentar que el tiempo de obtención del término n-ésimo $t(n)$ tiende a verificar también la sucesión de Fibonacci: $t(n) \rightarrow t(n-1) + t(n-2)$ cuando $n \rightarrow \infty$.

Práctica 3: Medición experimental de tiempos de ejecución en programas

```
C:\cygwin\bin\sh.exe
n:16 f(n):1597 ini.:31 fin:31 tiempo:0.0000
n:17 f(n):2584 ini.:31 fin:31 tiempo:0.0000
n:18 f(n):4181 ini.:31 fin:31 tiempo:0.0000
n:19 f(n):6765 ini.:31 fin:31 tiempo:0.0000
n:20 f(n):10946 ini.:31 fin:31 tiempo:0.0000
n:21 f(n):17711 ini.:31 fin:31 tiempo:0.0000
n:22 f(n):28657 ini.:31 fin:31 tiempo:0.0000
n:23 f(n):46368 ini.:31 fin:31 tiempo:0.0000
n:24 f(n):75025 ini.:46 fin:31 tiempo:0.0150
n:25 f(n):121393 ini.:46 fin:46 tiempo:0.0000
n:26 f(n):196418 ini.:62 fin:46 tiempo:0.0160
n:27 f(n):317811 ini.:78 fin:62 tiempo:0.0160
n:28 f(n):514229 ini.:93 fin:78 tiempo:0.0150
n:29 f(n):832040 ini.:124 fin:93 tiempo:0.0310
n:30 f(n):1346269 ini.:171 fin:124 tiempo:0.0470
n:31 f(n):2178309 ini.:249 fin:171 tiempo:0.0780
n:32 f(n):3524578 ini.:343 fin:249 tiempo:0.0940
n:33 f(n):5702887 ini.:421 fin:343 tiempo:0.0780
n:34 f(n):9227465 ini.:530 fin:421 tiempo:0.1090
n:35 f(n):14930352 ini.:748 fin:530 tiempo:0.2180
n:36 f(n):24157817 ini.:1092 fin:748 tiempo:0.3440
n:37 f(n):39088169 ini.:1604 fin:1092 tiempo:0.5920
n:38 f(n):63245986 ini.:2605 fin:1604 tiempo:0.9210
n:39 f(n):102334155 ini.:4227 fin:2605 tiempo:1.6220
n:40 f(n):165580141 ini.:6801 fin:4227 tiempo:2.5740
n:41 f(n):267914296 ini.:11091 fin:6801 tiempo:4.2900
n:42 f(n):433494437 ini.:18095 fin:11091 tiempo:7.0040
n:43 f(n):701408733 ini.:28922 fin:18095 tiempo:10.8270
n:44 f(n):1134903170 ini.:46674 fin:28922 tiempo:17.7520
```

El programa genera un fichero de nombre “**tiemposFibonacciRecursivo.txt**” con dos columnas de datos que son respectivamente **n** y **t(n)**. Podéis visualizarlos gráficamente desde Matlab ejecutando el programa **repreTiemposFibo.m**.



Para valorar la importancia de basar la implementación de los programas en algoritmos lo más eficientes posible, podéis experimentar ahora sobre el programa anterior, cambiando la implementación recursiva por esta otra iterativa:

Práctica 3: Medición experimental de tiempos de ejecución en programas

```
unsigned long fibonacciIterativo(unsigned char n){

    unsigned long a=1, b=1, c=-1, k=-1;

    for(k=2; k<=n; k++){
        c=a+b;
        a=b;
        b=c;
    }
    return b;
}
```

En particular, comparad el tiempo invertido en obtener los últimos términos de la sucesión entre las dos implementaciones propuestas: esta nueva implementación iterativa y la anterior implementación recursiva. ¿Cuál es la diferencia para los términos $n > 40$?

Medida experimental de tiempos para determinar todos los pares de números en el vector que sumen una cantidad dada

En el ejercicio vamos a implementar un programa que nos permita medir el tiempo de ejecución necesario para determinar todos los pares de números en el vector que sumen una cantidad dada. El programa trabajará con diversos tamaños de vector (generando en cada caso un vector de un tamaño dado, haciendo el proceso de búsqueda de pares y liberando posteriormente el vector). Cada vector se gestionará mediante un puntero a entero (`int *`) y se generarán progresivamente en el programa vectores cada vez más grandes para ver el crecimiento en tiempo computacional.

Para cada valor de tamaño n , el programa medirá el tiempo de ejecución $t(n)$ consumido en realizar la búsqueda de pares. La búsqueda de todos los pares (y correspondiente impresión por pantalla) la deberéis hacer en una función en el programa principal que recibe el vector y la cantidad que tienen que sumar los pares y se ocupe de buscar en el vector todos los pares que sumen esa cantidad (escribiéndolos por pantalla).

Adicionalmente debéis añadir la siguiente función al programa:

Práctica 3: Medición experimental de tiempos de ejecución en programas

- **void inicializaVectorRand(int *v1)**, que inicializa el vector con números aleatorios (utilizad srand y rand). **Los números aleatorios a introducir en el vector deben estar entre 0 y 100.** Esta función debe ser invocada tras generar el vector y antes de realizar el proceso de búsqueda de pares.

En el módulo principal nos encargaremos de realizar la experimentación. Para ello debéis implementar un programa que realice las siguientes tareas:

Para cada vector **v** de tamaño **n** (podéis tomar valores de **n=10.000**, ..., **100.000** con paso **10.000**; u otros tamaños, en función de lo que tarde en vuestros equipos, se trata de que los casos más lentos lleven por lo menos varios minutos):

- Crear el vector del tamaño requerido (pidiendo memoria con malloc).
- Llenar el vector con números aleatorios entre 0 y 100 (con la función indicada arriba).
- Imprimir por pantalla los pares de números que sumen una cantidad dada (función creada también por vosotros).
- Medir el tiempo **t(n)** que tarda en realizarse la operación anterior (ojo, sólo la búsqueda de pares, sin contabilizar el proceso de generación de aleatorios).
- Guardar en un fichero dos columnas de datos, respectivamente **n** y **t(n)** (formato idéntico al del ejemplo de Fibonacci).
- Liberar el vector.

El programa recibirá como argumentos en la línea de entrada los valores de tamaño inicial, tamaño final y paso necesarios para realizar la experimentación. En el ejemplo anterior recibiría 10000 100000 10000 pero el programa debe poder aceptar cualquier entrada de estos tres valores para poder realizar experimentos con tamaños iniciales, finales y pasos distintos.

Una vez ejecutado el programa, podéis visualizar una gráfica con los resultados **t(n)** frente a **n** utilizando el programa Matlab **repreTiemposPares.m**. Ejecutad el programa para distintos rangos de tamaños de vectores y analizad con cuidado las tendencias obtenidas. **El programa Matlab es únicamente para mostrar la gráfica con los resultados que saca a fichero vuestro programa C (podéis utilizar cualquier otro programa como R o Excel para generar las gráficas).**

Práctica 3: Medición experimental de tiempos de ejecución en programas

Medida experimental de tiempos de ordenación rápida (quicksort) en un vector dinámico.

En el ejercicio vamos a implementar un programa que nos permita medir los tiempos de ejecución para la estrategia de ordenación Quicksort sobre un vector dinámico de tamaño variable. Nuevamente, el programa trabajará con diversos tamaños de vector (generando en cada caso un vector de un tamaño dado, haciendo el proceso de ordenación y liberando posteriormente el vector). Cada vector se gestionará mediante un puntero a entero (`int *`) y se generarán progresivamente en el programa vectores cada vez más grandes para ver el crecimiento en tiempo computacional.

Para cada valor de tamaño n , el programa medirá el tiempo de ejecución $t(n)$ consumido en realizar la ordenación quicksort. El método de ordenación Quicksort lo debéis implementar en C en una función del programa que recibe el vector y lo ordena. Para documentaros sobre este tema, podéis utilizar la web <https://www.programiz.com/dsa/quick-sort> y obtener de ahí la implementación en C del Quicksort.

Adicionalmente debéis añadir la siguiente función al programa:

- **`void inicializaVectorRand(int *v1)`**, que inicializa el vector con números aleatorios (utilizad `srand` y `rand`). **Los números aleatorios a introducir en el vector deben estar entre 0 y 10000.** Esta función debe ser invocada tras generar el vector y antes de realizar el proceso de ordenación.

Esta ordenación tiene una complejidad temporal promedio de orden superlineal ($n \cdot \log(n)$).

En el módulo principal nos encargaremos de realizar la experimentación. Para ello debéis implementar un programa que realice las siguientes tareas:

Para cada vector v de tamaño n (podéis tomar valores de $n=10.000$, ..., 100.000 con paso 10.000 ; u otros tamaños, en función de lo que tarde en vuestros equipos, se trata de que los casos más lentos lleven por lo menos varios minutos):

- a. Crear el vector del tamaño requerido (pidiendo memoria con `malloc`)

Práctica 3: Medición experimental de tiempos de ejecución en programas

- b. Llenar el vector con números aleatorios entre 0 y 10000 (con la función indicada arriba)
- c. Ordenación quicksort
- d. Medir el tiempo $t(n)$ que tarda en realizarse la operación anterior (ojo, sólo la ordenación, sin contabilizar el proceso de generación de aleatorios).
- e. Guardar en un fichero dos columnas de datos, respectivamente n y $t(n)$ (formato idéntico al del ejemplo de Fibonacci).
- f. Liberar el vector

El programa recibirá como argumentos en la línea de entrada los valores de tamaño inicial, tamaño final y paso necesarios para realizar la experimentación. En el ejemplo anterior recibiría 10000 100000 10000 pero el programa debe poder aceptar cualquier entrada de estos tres valores para poder realizar experimentos con tamaños iniciales, finales y pasos distintos.

Una vez ejecutado el programa, podéis visualizar una gráfica con los resultados $t(n)$ frente a n utilizando el programa Matlab anterior u otro similar.

Medida experimental de tiempos de ordenación por el método de inserción.

Se trata de repetir lo anterior pero, en lugar de ordenar con Quicksort ordenaremos con el método de inserción. Este algoritmo tiene una complejidad temporal promedio de orden cuadrático (n^2). Su implementación en C la podéis obtener de:

<https://www.programiz.com/dsa/insertion-sort>

El ejercicio consiste en realizar la misma experimentación que en el ejercicio anterior, para obtener los tiempos de ejecución de la ordenación de sucesivos vectores mediante el método de inserción.

Una vez ejecutado el programa, podéis visualizar una gráfica con los resultados $t(n)$ frente a n .

IMPORTANTE: Dado que las tres experimentaciones (búsqueda de pares, Quicksort, Inserción) son muy similares, debéis optar por hacer un único programa que permita lanzar la experimentación de cualquiera de los tres casos (por ejemplo, poniendo el tipo de experimento a hacer como un argumento adicional en la línea de comandos, o bien pidiéndole al usuario el tipo de experimento a realizar al comenzar el programa).

Práctica 3: Medición experimental de tiempos de ejecución en programas

Entregables:

Se deberá realizar la entrega en el Campus Virtual. Las fechas de entrega se especifican en el Campus Virtual, y las instrucciones son las siguientes:

- En esta práctica hay que entregar **tanto el código del programa realizado como un informe explicativo** de la experimentación realizada:
 - Al igual que en prácticas anteriores, **el código a entregar estará únicamente compuesto por los archivos fuentes (*.c *.h) y un Makefile** que compile perfectamente. Cualquier ejercicio que no compile directamente con el makefile (sin opciones) será evaluado con la calificación de 0. Este criterio se mantendrá en el resto de prácticas de la asignatura.
 - El informe **debe ser entregado en formato PDF**. La entrega del documento en otros formatos tendrá automáticamente la calificación de 0. El informe será un breve documento (2 o 3 páginas) introduciendo el problema de experimentación, comentando los resultados, mostrando las tres gráficas y contestando a preguntas como:
 - ¿qué tamaños de vectores son gestionables en tiempo “razonable” por cada algoritmo? Podéis por ejemplo establecer un umbral máximo de tiempo razonable (varios minutos al menos) y a partir de ahí razonar a qué tamaños llega cada algoritmo poniendo ese límite máximo de tiempo.
 - ¿Es más eficiente en general alguna de las tres operaciones? En caso afirmativo, indica si es así para todos los valores de **n** o únicamente para algunos. En este último caso, indica el rango de valores de **n** para los que es más rápida cada operación.
 - El fichero comprimido conteniendo el código y el PDF tendrá el nombre **apellido1_apellido2** y la extensión **.ZIP (usad siempre .ZIP como formato de compresión)**.

Algunos errores típicos detectados en años anteriores y que debéis evitar a la hora de redactar el informe (**serán tenidos en cuenta en la revisión del informe**):

- gráficas caóticas por no probar rangos de tamaños suficientemente grande. No se aprecia crecimiento superlineal o cuadrático.
- errores edición informe (p.e. falta de justificación derecha de los párrafos, falta de nombre de autor en el informe, errores ortográficos, falta de título en el documento, falta de introducción, edición no profesional –tipos de fuente heterogéneos, etc-).

Práctica 3: Medición experimental de tiempos de ejecución en programas

- alguna de las preguntas planteadas en el guión no es contestada en el informe o de hacerlo se hace de manera genérica sin proporcionar datos experimentales.
- argumentar en el informe que superlineal=lineal, o que cuadrático=superlineal, o que cuadrático=exponencial (es decir, tener confusión respecto a las clases de complejidad).
- la cronometración no cuenta sólo el coste de ordenar o buscar los pares sino también alguna operación adicional (creación, relleno de aleatorios o liberación del vector, por ejemplo).