Practica 1

1. El algoritmo fib.c resuelve la serie de Fibonacci, para un número N dado, utilizando dos métodos: recursivo e iterativo. Analice los tiempos de ejecución de ambos métodos ¿Cuál es más rápido? ¿Por qué?
   Nota: ejecute con N=1..50.

   Ejecución:

   Calculando Fibonacci iterativo para n = 40. . .
    Resultado = 102334155
    Tiempo en segundos 0.0000005960
   Calculando Fibnacci recursivo para n = 40. . .
    Tiempo en segundos 0.5311213017

   Calculando Fibonacci iterativo para n = 50. . .
    Resultado = 12586269025
    Tiempo en segundos 0.0000002146
   Calculando Fibnacci recursivo para n = 50. . .
    Tiempo en segundos 64.0037203074

   El algoritmo más rápido es el iterativo dado que no hace llamados constantes a nuevas funciones con sus respectivos espacios de direcciones. Estas nuevas ejecuciones no serán paralelas dado que dependen completamente entre sí.

2. El algoritmo funcion.c resuelve, para un x dado, la siguiente sumatoria:

$$\sum_{i=0}^{100\,000\,000} 2 * \frac{x^3 + 3x^2 + 3x + 2}{x^2 + 1} - i$$

   El algoritmo compara dos alternativas de solución. ¿Cuál de las dos formas es más rápida? ¿Por qué?

   Ejecución:

   Function calculada...
    Tiempo total en segundos 0.2703552246
    Tiempo promedio en segundos 0.0000000027
   Función calculada cada vez
    Tiempo total en segundos 0.4145798683
    Tiempo promedio en segundos 0.0000000041

   El primer algoritmo es más rápido dado que realiza un menor número de operaciones en cada iteración, ya que guarda una parte de la formula en una variable ya calculada, mientras que el otro lo calcula en cada iteración.

3. El algoritmo instrucciones.c compara el tiempo de ejecución de las operaciones básicas: suma (+), resta (-), multiplicación (*) y división (/), para dos operandos dados x e y. ¿Qué análisis se puede hacer de cada operación? ¿Qué ocurre si x e y son potencias de 2?

   Ejecucion:

   Suma...
   Tiempo total en segundos 1.9997000694
   Tiempo promedio en segundos 0.0000000020
   Resta...
   Tiempo total en segundos 1.8734669685
   Tiempo promedio en segundos 0.0000000019
   Producto...
   Tiempo total en segundos 2.0060691833
   Tiempo promedio en segundos 0.0000000020
   División...
   Tiempo total en segundos 4.1427259445
   Tiempo promedio en segundos 0.0000000041

   La división es aquella que tardan más en ejecutarse dado que implican un uso de operaciones más básicas para llegar a su resultado porque no está implementada en hardware para (como son la suma, la resta y el producto)

   Ejecución con potencias de 2 (x = 1024, y =256):

   Suma...
   Tiempo total en segundos 1.9920659065
   Tiempo promedio en segundos 0.0000000020
   Resta...
   Tiempo total en segundos 1.8774650097
   Tiempo promedio en segundos 0.0000000019
   Producto...
   Tiempo total en segundos 1.9643180370
   Tiempo promedio en segundos 0.0000000020
   División...
   Tiempo total en segundos 2.3637049198
   Tiempo promedio en segundos 0.0000000024

   Si x e y son potencias de 2 la ejecución será más rápida en la división dado que el número de operaciones para realizarla disminuirá ya que la implementación en binario es más rápida.

4. En función del ejercicio anterior analice el algoritmo instrucciones2.c que resuelve una operación binaria (dos operandos) con dos operaciones distintas.

   División...
   Tiempo total en segundos 4.1434111595

Tiempo promedio en segundos 0.0000000041
Producto...
Tiempo total en segundos 1.8790352345
Tiempo promedio en segundos 0.0000000019
Resultado correcto

El programa realiza la misma fórmula pero con dos operaciones distintas, la multiplicación y la división. Este nos muestra que siempre es mejor multiplicar un numero por un decimal antes que dividirlo por su entero equivalente, ya que la ejecución será más rápida.

5. Investigue en la documentación del compilador o a través de Internet qué opciones de optimización ofrece el compilador gcc (flag O). Compile y ejecute el algoritmo matrices.c, el cual resuelve una multiplicación de matrices de NxN. Explore los diferentes niveles de optimización para distintos tamaños de matrices. ¿Qué optimizaciones aplica el compilador? ¿Cuál es la ganancia respecto a la versión sin optimización del compilador? ¿Cuál es la ganancia entre los distintos niveles?

   -O0: Este nivel (que consiste en la letra "O" seguida de un cero) desconecta por completo la optimización y es el predeterminado si no se especifica ningún nivel -O en CFLAGS o CXXFLAGS. El código no se optimizará. Esto, normalmente, no es lo que se desea.
   -O1: El nivel de optimización más básico. El compilador intentará producir un código rápido y pequeño sin tomar mucho tiempo de compilación. Es básico, pero conseguirá realizar correctamente el trabajo.
   -O2: Un paso delante de -O1. Es el nivel recomendado de optimización, a no ser que el sistema tenga necesidades especiales. -O2 activará algunas opciones añadidas a las que se activan con -O1. Con -O2, el compilador intentará aumentar el rendimiento del código sin comprometer el tamaño y sin tomar mucho más tiempo de compilación. Se puede utilizar SSE o AVX en este nivel pero no se utilizarán registros YMM a menos que también se habilite ftree-vectorize.
   -O3: El nivel más alto de optimización posible. Activas optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. El hecho de compilar con -O3 no garantiza una forma de mejorar el rendimiento y, de hecho, en muchos casos puede ralentizar un sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria. También se sabe que -O3 puede romper algunos paquetes. No se recomienda utilizar -O3. Sin embargo, también habilita -ftree-vectorize de modo que los bucles dentro del código se vectorizarán y se utilizarán los registros AVX YMM.
   -Ofast: Nuevo en GCC 4.7. Consiste en el ajuste -O3 más las opciones -ffast-math, -fno-protect-parens y -fstack-arrays. Esta opción rompe el cumplimiento de estándares estrictos y no se recomienda su utilización.
   -Os: Optimizará el tamaño del código. Activa todas las opciones de -O2 que no incrementan el tamaño del código generado. Es útil para máquinas con capacidad limitada de disco o con CPUs que tienen poca caché.
   -Og: En GCC 4.8 aparece un nuevo nivel de la optimización general: -Og. Trata de solucionar la necesidad de realizar compilaciones más rápidas y obtener una

experiencia superior en la depuración a la vez que ofrece un nivel razonable de rendimiento en la ejecución. La experiencia global en el desarrollo debería ser mejor que para el nivel de optimización -O0. Observe que -Og no implica -g, éste simplemente deshabilita optimizaciones que podrían interferir con la depuración.

Ejecucion:

O0
Multiplicación de matrices de 500x500. Tiempo en segundos 1.866930
Multiplicación de matrices resultado correcto
O1
Multiplicación de matrices de 500x500. Tiempo en segundos 0.124592
Multiplicación de matrices resultado correcto
O2
Multiplicación de matrices de 500x500. Tiempo en segundos 0.123973
Multiplicación de matrices resultado correcto
O3
Multiplicación de matrices de 500x500. Tiempo en segundos 0.121252
Multiplicación de matrices resultado correcto
Ofast
Multiplicación de matrices de 500x500. Tiempo en segundos 0.102309
Multiplicación de matrices resultado correcto
Os
Multiplicación de matrices de 500x500. Tiempo en segundos 0.123857
Multiplicación de matrices resultado correcto
Og
Multiplicación de matrices de 500x500. Tiempo en segundos 0.841110
Multiplicación de matrices resultado correcto

El compilador por defecto ejecuta con la flag -O0, que no realiza ninguna optimización. Las ganancias son vistas en el tiempo de ejecución obtenido en la parte superior.

6. Dada la ecuación cuadrática: $x^2 - 4.0000000\ x + 3.9999999 = 0$, sus raíces son r1 = 2.000316228 y r2 = 1.999683772 (empleando 10 dígitos para la parte decimal).

    a. El algoritmo quadratic1.c computa las raíces de esta ecuación empleando los tipos de datos float y double. Compile y ejecute el código. ¿Qué diferencia nota en el resultado?

    Ejecucion:

    Soluciones Float: 2.00000 2.00000
    Soluciones Double: 2.00 032 1.99968

    La solución que utiliza el tipo de variable double es la más cercana a las verdaderas raíces, ya que utiliza más bits de precisión.

b. El algoritmo quadratic2.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución?

   Sin optimización:

   Ejecucion TIMES 100:

   Tiempo requerido solución Double: 34.826516
   Tiempo requerido solución Float: 39.888028

   Ejecucion TIMES 50:

   Tiempo requerido solución Double: 17.385123
   Tiempo requerido solución Float: 19.961462

   Ejecucion TIMES 10:

   Tiempo requerido solución Double: 3.454544
   Tiempo requerido solución Float: 3.998400

   Con Ofast:

   Ejecucion TIMES 100:

   Tiempo requerido solución Double: 2.413033
   Tiempo requerido solución Float: 1.993287

   Ejecucion TIMES 50:

   Tiempo requerido solución Double: 1.153039
   Tiempo requerido solución Float: 1.010707

   Ejecucion TIMES 10:

   Tiempo requerido solución Double: 0.000000
   Tiempo requerido solución Float: 0.000000

   La ejecución con variables double sin optimizaciones en compilación tardaran menos en ejecutarse ya que la implementación de la arquitectura puede tener mejoras para este tipo (como es la maquina donde se ejecutó este ejemplo), pero el resultado genérico (o con optimizaciones al compilar) debería dar a float como el más rápido ya que es el que posee menos bits. A medida que se reduce las iteraciones el tiempo de diferencia disminuirá.

c. El algoritmo quadratic3.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES.

¿Qué diferencia nota en la ejecución? ¿Qué diferencias puede observar en el código con respecto a quadratic2.c?

Ejecucion Times 100:

Tiempo requerido solución Double: 54.636505
Tiempo requerido solución Float: 37.775923

Ejecucion Times 50:

Tiempo requerido solución Double: 25.801075
Tiempo requerido solución Float: 18.557679

Ejecucion Times 10:

Tiempo requerido solución Double: 4.861281
Tiempo requerido solución Float: 3.562021

La diferencia en ejecución es que la ventaja de float sobre double es notable aun sin ninguna optimización en el compilador.
La diferencia en el código es que los números flotantes están declarados explícitamente como float, de esta manera el compilador puede utilizar optimizaciones sin ninguna flag activa.

7. Analice los algoritmos iterstruc1.c e iterstruc2.c que resuelven una multiplicación de matrices utilizando dos estructuras de control distintas. ¿Cuál de las dos estructuras de control tiende a acelerar el cómputo? Compile con y sin opciones de optimización del compilador

   Ejecución Sin Optimización:

   Inicializando matrices...
   Calculando For...
   Tiempo For en segundos 6.034467

   Inicializando matrices...
   Calculando While...
   Tiempo While en segundos 5.900395

   Ejecución Con Optimización Ofast:

   Inicializando matrices...
   Calculando For...
   Tiempo For en segundos 0.898732

   Inicializando matrices...
   Calculando While...
   Tiempo While en segundos 0.885625

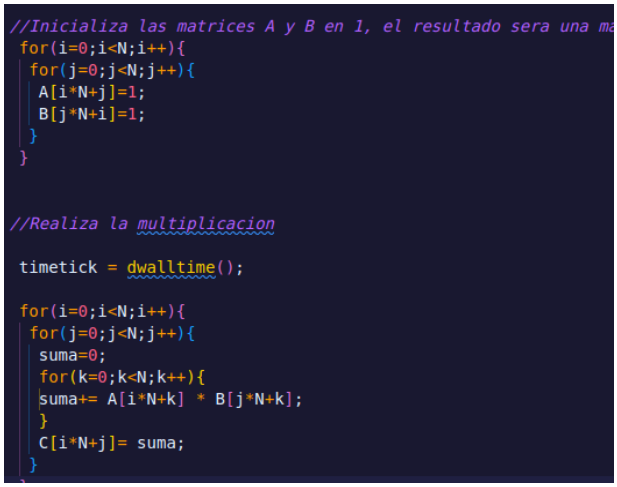
Las dos estructuras tienen similar tiempo de ejecución y se alternan aleatoriamente cual es la más rápida.

8. Analice el algoritmo matrices.c. ¿Dónde cree que se producen demoras? ¿Cómo podría optimizarse el código? Al menos, considere los siguientes aspectos:

   - Explotación de localidad de datos a través de reorganización interna de matrices A, B o C (según corresponda).
   - El uso de Setters y getters es una buena práctica en la programación orientada a objetos. ¿Tiene sentido usarlos en este caso? ¿cuál es su impacto en el rendimiento?
   - ¿Hay expresiones en el cómputo que pueden refactorizarse para no ser computadas en forma repetida?
   - En lugar de ir acumulando directamente sobre la posición C[i,j] de la matriz resultado (línea 72), pruebe usar una variable local individual y al finalizar el bucle más interno, asigne su valor a C[i,j]. ¿Esta modificación impacta en el rendimiento? ¿Por qué?

   Combine las mejoras que haya encontrado para obtener una solución optimizada y compare los tiempos con la solución original para diferentes tamaños de matrices.

   - Con respecto a la optimización por localidad de datos, cómo se accede para la multiplicación a la matriz B por columnas, se aprovecharía mejor ordenando a la misma por columnas y haciendo un acceso a ella de dicha manera.



```
//Inicializa las matrices A y B en 1, el resultado sera una ma
 for(i=0;i<N;i++){
  for(j=0;j<N;j++){
   A[i*N+j]=1;
   B[j*N+i]=1;
  }
 }


//Realiza la multiplicacion

 timetick = dwalltime();

 for(i=0;i<N;i++){
  for(j=0;j<N;j++){
   suma=0;
   for(k=0;k<N;k++){
   suma+= A[i*N+k] * B[j*N+k];
   }
   C[i*N+j]= suma;
  }
```

   - Los getters y setters son innecesarios ya que no utilizamos el concepto de clases ni el polimorfismo dado que la orientación de la matriz es predefinida en el código. Esto evita múltiples llamados a funciones que ralentizan la ejecución.
   - No existen expresiones que puedan ser refactorizadas para no ser computadas.

- La modificación de utilizar una variable auxiliar para ir guardando la suma de la posición tiene un impacto sobre el rendimiento, esto dado a que se almacena en un registro en vez de acceder repetidas veces a la memoria principal a buscar los valores que se tienen en el vector C (hay menor latencia).

Tiempos de ejecución:

Sin optimización:
Multiplicación de matrices de 1000x1000. Tiempo en segundos 24.325247
Multiplicación de matrices resultado correcto

Con optimización, pero sin uso de variable auxiliar:
Multiplicación de matrices de 1000x1000. Tiempo en segundos 4.711709
Multiplicación de matrices resultado correcto

Con optimización, pero con uso de variable auxiliar:
Multiplicación de matrices de 1000x1000. Tiempo en segundos 3.713192
Multiplicación de matrices resultado correcto

9. Analice y describa brevemente cómo funciona el algoritmo mmblk.c que resuelve la multiplicación de matrices cuadradas de NxN utilizando una técnica de multiplicación por bloques. Luego, ejecute el algoritmo utilizando distintos tamaños de matrices y distintos tamaños de bloque (pruebe con valores que sean potencia de 2; p.e. N={512,1024,2048} y TB={16,32,64,128}). Finalmente, compare los tiempos con respecto a la multiplicación de matrices optimizada del ejercicio anterior. Según el tamaño de las matrices y de bloque elegido, responda: ¿Cuál es más rápido? ¿Por qué? ¿Cuál sería el tamaño de bloque óptimo para un determinado tamaño de matriz? ¿De qué depende el tamaño de bloque óptimo para un sistema?

   La multiplicación de matrices por bloques es una técnica que divide las matrices en submatrices o bloques más pequeños, lo que permite que la multiplicación se realice en estas partes más pequeñas. El algoritmo funciona de la siguiente manera:
   - Divide cada matriz (NxN) en bloques de tamaño (TBxTB).
   - Multiplica los bloques correspondientes de las dos matrices.
   - Suma los resultados para obtener los bloques de la matriz final.

   https://www.youtube.com/watch?v=OSelhO6Qnlc

   Ejecución:

   Con N = 512
   Multiplicación de matrices de 512x512. Tiempo en segundos 0.509872
   Multiplicación de matrices resultado correcto

   MMBLK-SEC;512;16;0.755029;0.355530

Con N = 1024
Multiplicación de matrices de 1024x1024. Tiempo en segundos 3.713192
Multiplicación de matrices resultado correcto

MMBLK-SEC;1024;16;5.930494;0.362109

A mayor tamaño de matriz más notoria la diferencia de velocidad, dándole ventaja a la multiplicación por bloque, ya que la relación entre tamaño de bloque y tamaño de matriz es mayor por lo tanto se justifica el computo extra que trae la división.
La multiplicación de matrices por bloques puede ser más eficiente que la multiplicación de matrices estándar, especialmente cuando se trabaja con matrices grandes, ya que puede adaptarse mejor a la jerarquía de memoria de una computadora y aprovechar mejor la localidad de datos.
El tamaño de bloque óptimo para un determinado tamaño de matriz dependerá del tamaño de la caché y cómo se mapean los datos en ella. Un bloque pequeño podría no aprovechar toda la caché disponible, mientras que un bloque demasiado grande podría causar fallos de caché.
El tamaño de bloque óptimo también depende de:

- La arquitectura del sistema (por ejemplo, tamaño de caché, velocidad de acceso a memoria).
- El patrón de acceso a los datos durante la multiplicación.
- La implementación específica del algoritmo de multiplicación por bloques.

10. Analice el algoritmo triangular.c que resuelve la multiplicación de una matriz cuadrada por una matriz triangular inferior, ambas de NxN. ¿Cómo se podría optimizar el código? ¿Se pueden evitar operaciones? ¿Se puede reducir la memoria reservada? Implemente una solución optimizada y compare los tiempos probando con diferentes tamaños de matrices.

    Se podría utilizar una variable auxiliar para reducir tiempo de acceso y además no calcular aquellas posiciones donde la matriz triangular inferior tiene 0 ya que siempre resultaran en 0.

    Ejecución sin optimización:
    Tiempo en segundos 5.494969
    Multiplicación de matriz triangular correcta

    Ejecución con optimización:
    Tiempo en segundos 5.038927
    Multiplicación de matriz triangular correcta