UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

Programación II
Práctica Calificada 2
Pregrado
2025-II
Profesor: Rubén Rivas
Lab: 2.03

Indicaciones específicas:

- Esta evaluación contiene 9 páginas (incluyendo esta página) con 3 preguntas. El total de puntos son 20.
- El tiempo límite para la evaluación es 100 minutos.
- Cada pregunta deberá ser respondida en su respectiva pregunta siguiendo el formato:
  - `p1.cpp`, `p1.h`
  - `p2.cpp`, `p2.h`
  - `p3.cpp`, `p3.h`
- Deberás subir estos archivos directamente a www.gradescope.com, uno en cada ejercicio. También puedes crear un .zip
- La evaluación es **individual**. Un nivel alto de **similitud** con otros estudiantes o fuentes externas no será aceptada y se **anulará** el ejercicio.
- Se puede **consultar material de clase** y utilizar funciones o partes de código desarrollados en clase. Esto ultimo no descontará puntos, pero se debe **hacer referencia a ellos en la entrega**.

Competencias:

- Para los alumnos de la carrera de Ciencia de la Computación

  Aplicar conocimientos de computación apropiados para la solución de problemas definidos y sus requerimientos en la disciplina del programa. (nivel 2)

  Diseñar, implementar y evaluar soluciones a problemas complejos de computación.(nivel 2)

  Crear, seleccionar, adaptar y aplicar técnicas, recursos y herramientas modernas para la práctica de la computación y comprende sus limitaciones. (nivel 2)

- Para los alumnos de las carreras de Ingeniería

  Aplicar conocimientos de ingeniería en la solución de problemas complejos de ingeniería (nivel 2).

  Diseñar soluciones relacionados a problemas complejos de ingeniería (nivel 2)

Crear, seleccionar y utilizar técnicas, habilidades, recursos y herramientas modernas de la ingeniería y las tecnologías de la información, incluyendo la predicción y el modelamiento, con la comprensión de sus limitaciones (nivel 2)

- Para los alumnos de Administración y Negocios Digitales

  Analizar información verbal y/o lógica proveniente de distintas fuentes, encontrando relaciones y presentándola de manera clara y concisa (nivel 2)

  Analizar y evaluar el comportamiento del consumidor y el desarrollo de estrategias comerciales (nivel 2)

  Trabajar de manera efectiva con equipos multidisciplinarios y diversos en género, nacionalidad, edad, etc. (nivel 2)

Calificación:

Tabla de puntos (sólo para uso del professor)

Question	Points	Score
1	7	
2	6	
3	7	
Total:	20	

1. (7 puntos) **Matriz Dinámica y Recorrido en Espiral**

   Implemente un conjunto de **funciones** para gestionar una matriz $n \times n$ de enteros en memoria dinámica y para imprimir su **recorrido en espiral** iniciando por la **primera columna hacia abajo**, luego borde inferior hacia la derecha, borde derecho hacia arriba y borde superior hacia la izquierda, repitiendo el patrón hasta cubrir toda la matriz.

   Interfaz sugerida (procedural):

   - `int** crear_matriz(int n)`: reserva memoria para una matriz $n \times n$.
   - `void liberar_matriz(int**& M, int n)`: libera toda la memoria de la matriz.
   - `void leer_matriz(int** M, int n)`: carga los elementos desde la entrada estándar.
   - `void recorrido_espiral(int** M, int n)`: imprime los elementos en el orden espiral descrito, en una sola línea separados por espacio.

   Requisitos mínimos:

   - Validar $n \geq 1$ y evitar accesos fuera de rango.
   - Asegurar que no existan fugas de memoria (la función de liberación debe ser correcta).
   - El formato de salida del recorrido debe ser una única línea con enteros separados por espacio.

   Casos de prueba:

   T1 $n = 2$; $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ $\Rightarrow$ `1 3 4 2`

   T2 $n = 3$; $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $\Rightarrow$ `1 4 7 8 9 6 3 2 5`

   T3 $n = 1$; $[\,42\,]$ $\Rightarrow$ `42`

   T4 $n = 4$; $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$ $\Rightarrow$ `1 5 9 13 14 15 16 12 8 4 3 2 6 10 11 7`

Criterio	Excelente	Adecuado	Mínimo	Insuficiente
Algoritmo: Evalúa el diseño del algoritmo, siguiendo buenas prácticas en programación. Asi como la ejecución del mismo	El diseño del algoritmo es ordenado y claro, siguiendo buenas prácticas en programación. La ejecución es correcta (3pts)	El diseño del algoritmo es ordenado y claro, pero optimizable. La ejecución es correcta (2pts)	El diseño del algoritmo contiene algunos errores que afectan la ejecución (1pts).	El diseño del algoritmo y la ejecución son incorrectos (0 pts)
Código : Evalúa sintaxis en el código y correcta ejecución (semántica)	No contiene errores sintácticos o de compilación. La ejecución es correcta (2pts)	Existen algunos errores sintácticos, que no afectan directamente el resultado, pero hacen al código optimizable. (1.5pts).	Existen errores sintácticos o de ejecución, que afectan parcialmente el resultado (1pts).	El código tiene errores de sintáxis y de ejecución que no permiten obtener un resultado correcto (0 pts).
Eficiencia: evalua uso de buenas practicas en programación en el diseño del algoritmo y el código de programación, para lograr un nivel de eficiencia adecuado	El código es óptimo y eficiente. De buen performance e interacción con el usuario (2 pts)	El codigo es de buen performance durante la ejecución pero optimizable. Pero no afecta el resultado. (1.5 pts).	El código no esta optimizado, lo que afecta parcialmente el resultado. (0.5pts).	El codigo no esta optimizado y la ejecución es deficiente (0pts).

2. (6 puntos) **Verificación de Subconjunto usando vectores**

   Implemente una función `es_subconjunto` que, dadas dos listas de `strings`, `base` y `consulta`, verifique si cada elemento de `consulta` pertenece a `base`. La función retornará `true` cuando la lista de consulta sea un **subconjunto** de la lista base y `false` en caso contrario.

```
bool es_subconjunto(vector<string> base, vector<string> consulta);
```

   Casos de prueba:

   T1 Base: {"a", "b", "c"}; Consulta: {"a"} ⇒ `true`
   T2 Base: {"peru", "lima", "arequipa"}; Consulta: {"lima", "cusco"} ⇒ `false`
   T3 Base: {"x", "x", "y"}; Consulta: {"x", "y"} ⇒ `true`
   T4 Base: {}; Consulta: {"algo"} ⇒ `false`

Los criterios en la rúbrica (y el puntaje respectivo) se condicionan a que la solución presentada corresponda al problema planteado. La rúbrica para esta pregunta es:

Criterio	Excelente	Adecuado	Mínimo	Insuficiente
Algoritmo: Evalúa el diseño del algoritmo, siguiendo buenas prácticas en programación. Asi como la ejecución del mismo	El diseño del algoritmo es ordenado y claro, siguiendo buenas prácticas en programación. La ejecución es correcta (`3pts`)	El diseño del algoritmo es ordenado y claro, pero optimizable. La ejecución es correcta (`2pts`)	El diseño del algoritmo contiene algunos errores que afectan la ejecución (`1pts`).	El diseño del algoritmo y la ejecución son incorrectos (`0 pts`)
Código : Evalúa sintaxis en el código y correcta ejecución (semántica)	No contiene errores sintácticos o de compilación. La ejecución es correcta (`2pts`)	Existen algunos errores sintácticos, que no afectan directamente el resultado, pero hacen al código optimizable. (`1.5pts`).	Existen errores sintácticos o de ejecución, que afectan parcialmente el resultado (`1pts`).	El código tiene errores de sintáxis y de ejecución que no permiten obtener un resultado correcto (`0 pts`).
Eficiencia: evalua uso de buenas practicas en programación en el diseño del algoritmo y el código de programación, para lograr un nivel de eficiencia adecuado	El código es óptimo y eficiente. De buen performance e interacción con el usuario (`1pt`)	El codigo es de buen performance durante la ejecución pero optimizable. Pero no afecta el resultado. (`0.7pts`).	El código no esta optimizado, lo que afecta parcialmente el resultado. (`0.3pts`).	El codigo no esta optimizado y la ejecución es deficiente (`0pts`).

3. (7 puntos) **Simulador MonteCarloPI - Clases y Objetos**

   Implemente una clase `class MonteCarloPISimulator` que estime el valor de π mediante el método de *Monte Carlo*. La simulación genera puntos aleatorios (x, y) uniformes en el cuadrado $[0, 1]^2$ y cuenta cuántos de ellos cumplen $x^2 + y^2 \leq 1$. La estimación final se calcula como:

   $$\widehat{\pi} = 4 \cdot \frac{\text{aciertos}}{N},$$

   donde N es el número total de puntos generados.

   Interfaz sugerida:

   - `MonteCarloPISimulator(int seed)`: constructor que inicializa el generador aleatorio con una semilla determinada (para reproducir resultados).
   - `bool set_iterations(int N)`: define el número de puntos que se generarán en la siguiente ejecución; retorna `false` si $N = 0$.
   - `bool run()`: ejecuta una corrida de N puntos y acumula los resultados; retorna `false` si no se ha configurado un N válido.
   - `bool add_batch(int k)`: agrega k puntos adicionales al total acumulado.
   - `double pi_hat()const`: devuelve la estimación actual de π si existen datos acumulados; de lo contrario, retorna 0.
   - `double abs_error()`: error absoluto $|\widehat{\pi} - \pi|$ respecto al valor real de π.
   - `double rel_error()`: error relativo $|\widehat{\pi} - \pi|/\pi$.
   - `void reset()`: reinicia los contadores internos sin cambiar la semilla.

   Nota: Todos los métodos de ejecución deben validar sus parámetros y retornar un valor lógico que indique éxito o error, sin usar excepciones.

   Función para generar números aleatorios de 0 a 1:

```cpp
#include <random>

double uniform01_once(unsigned int seed) {
  std::mt19937 rng(seed);  // engine sembrado con 'seed'
  std::uniform_real_distribution<double> U(0.0, 1.0);
  return U(rng);           // un valor en [0,1)
}
```

Casos de prueba:

T1 **Corrida básica e impresión de métricas**

```
// Corrida básica (N=10,000)
MonteCarloPISimulator sim(42);
if (sim.set_iterations(10000) && sim.run()) {
  std::cout
    << "piHat="
    << sim.pi_hat()
    << "\n"; // ~3.14
}
```

T2 **Ejecucion en lotes (acumulativa)**

```
// Ejecución en lotes acumulativos (20k + 30k)
MonteCarloPISimulator sim(2025);
bool ok = sim.add_batch(20000) && sim.add_batch(30000) && sim.run
   ();
if (ok) {
  std::cout
    << "piHat=" << sim.pi_hat()
    << "\n"; // cercano a 3.14
}
```

T3 **Reproducibilidad con la misma semilla**

```
// Reproducibilidad (misma semilla y N => mismo resultado)
MonteCarloPISimulator a(7), b(7);
a.set_iterations(1000); a.run();
b.set_iterations(1000); b.run();
std::cout
  << (std::abs(a.pi_hat() - b.pi_hat()) < 1e-12 ? "OK" : "DIF")
  << "\n";
```

T4 **Entradas invalidas y control por retorno**

```
// Control de errores sin excepciones
MonteCarloPISimulator sim(123);
if (!sim.set_iterations(0)) {
  std::cout << "N invalido\n"; // manejo por retorno booleano
}
sim.reset();                   // deja total=0
if (sim.pi_hat() == 0.0) {
  std::cout << "Sin datos\n";  // pi_hat no válido sin muestras
}
```

Los criterios en la rúbrica (y el puntaje respectivo) se condicionan a que la solución presentada corresponda al problema planteado. La rúbrica para esta pregunta es:

Criterio	**Excelente**	**Adecuado**	**Mínimo**	**Insuficiente**
Algoritmo: Evalúa el diseño del algoritmo, siguiendo buenas prácticas en programación. Asi como la ejecución del mismo	El diseño del algoritmo es ordenado y claro, siguiendo buenas prácticas en programación. La ejecución es correcta (`3pts`)	El diseño del algoritmo es ordenado y claro, pero optimizable. La ejecución es correcta (`2pts`)	El diseño del algoritmo contiene algunos errores que afectan la ejecución (`1pts`).	El diseño del algoritmo y la ejecución son incorrectos (`0 pts`)
Código : Evalúa sintaxis en el código y correcta ejecución (semántica)	No contiene errores sintácticos o de compilación. La ejecución es correcta (`2pts`)	Existen algunos errores sintácticos, que no afectan directamente el resultado, pero hacen al código optimizable. (`1.5pts`).	Existen errores sintácticos o de ejecución, que afectan parcialmente el resultado (`1pts`).	El código tiene errores de sintáxis y de ejecución que no permiten obtener un resultado correcto (`0 pts`).
Eficiencia: evalua uso de buenas practicas en programación en el diseño del algoritmo y el código de programación, para lograr un nivel de eficiencia adecuado	El código es óptimo y eficiente. De buen performance e interacción con el usuario (`2 pts`)	El codigo es de buen performance durante la ejecución pero optimizable. Pero no afecta el resultado. (`1.5 pts`).	El código no esta optimizado, lo que afecta parcialmente el resultado. (`0.5pts`).	El codigo no esta optimizado y la ejecución es deficiente (`0pts`).