Universidad de Granada
Departamento de Ciencias de la Computación
e Inteligencia Artificial

Fundamentos de Programación (2019/2020)
1º GII / GII-M / GII-ADE
Conv. Ordinaria - 8 de Enero de 2020

DECSAI

Normas para la realización del examen: **Duración: 2.5 horas**

- **Debe disponer de un documento oficial que acredite su identidad a disposición del profesor.**
- **No olvide escribir su nombre completo y grupo en todos y cada uno de los folios que entregue.**

◁ Ejercicio 1 ▷ Robot [3 puntos]

Se dispone de un robot que se mueve en una línea, utilizando pasos de longitud fija. El robot se ubica en una posición `pos`, siendo `pos` un valor entero $1 \leq$ `pos` ≤ 100. Luego, el robot ejecuta una serie de órdenes, indicadas mediante un vector `ordenes` de tipo char, con longitud `longitud`. Cada orden es una letra 'I' o 'D', indicando si el robot se mueve a la izquierda (decrementando la posición actual `pos` en 1 unidad) o a la derecha (incrementando la posición actual `pos` en 1 unidad). Si es una letra distinta, el robot permanecerá en la misma posición.

Las posiciones **válidas** del robot cumplen $1 \leq$ `pos` ≤ 100. Se dice que **una serie de órdenes es correcta si el robot nunca se sale de las posiciones válidas**.

Se pide implementar un programa (directamente en la función `main`) para que, dada una posición inicial `pos_inicial` y un vector de órdenes de longitud `longitud`, haga lo siguiente:

- Si la serie de órdenes es correcta, muestre cuántas veces se visitó cada posición.
- Si la serie de órdenes NO es correcta, el programa terminará indicando cuántas órdenes se pudieron ejecutar.

No hace falta que escriba el código de programa que lee/calcula los valores de `pos_inicial`, `ordenes` y `longitud`.

Ejemplos

`pos_inicial: 10`
`longitud: 6`
`ordenes: DDIIII`
 	
Posiciones (válidas) visitadas:
10→11→12→11→10→9→8
Salida del programa
Serie de órdenes: correcta
Frecuencia de visitas por posición:
(8,1), (9,1), (10,2), (11,2), (12,1).
	`pos_inicial: 1`
	`longitud: 4`
	`ordenes: DIID`
	
	Posiciones (válidas) visitadas:
	1→2→1
	Salida del programa
	Serie de órdenes: incorrecta.
	Se ejecutaron 2 órdenes.

◁ Ejercicio 2 ▷ Relieve [4 puntos]

El relieve de una región geográfica se puede representar mediante una tabla rectangular `alt`, donde cada elemento `alt[fil][col]` representa la altura (sobre el nivel del mar), de la parcela `(fil,col)` del terreno (`fil` y `col` son enteros). Una parcela se define como "pico" si sus 8 parcelas vecinas tienen una altura menor. Por simplicidad, se supone que las parcelas de los bordes no pueden ser picos. Se propone la representación para la clase `Relieve` mostrada en la tabla 1. Remarcar que **todos los objetos de esta clase estarán registrados respecto a la coordenada** $(0, 0)$.

Relieve
- static const int NUM_FILAS = 100 - static const int NUM_COLS = 100 - int alt[NUM_FILAS][NUM_COLS] - int filas_utilizadas, cols_utilizadas
+ Relieve(int num_filas, int num_cols) + int FilasUtilizadas() + int ColumnasUtilizadas() + int Altura(int fil, int col) + void SetAltura(int fil, int col, int altura)

Table 1: Propuesta para la clase `Relieve`

Implemente los siguientes métodos:

1. (0.75) `EsPico`: Devuelve `true` si la parcela situada en las coordenadas `(fil,col)`, es un pico.
2. (0.5) `ObtenerPicos`: Construye y devuelve un objeto de la `clase SecuenciaPuntos` con las coordenadas de las parcelas que son picos.
3. (0.75) `Fusion`: Involucra dos objetos de la clase `Relieve`. Construye y devuelve un nuevo objeto con:
   - El número de filas/columnas útiles del nuevo objeto será el mínimo entre el número de filas/columnas útiles de los objetos involucrados.
   - Cada parcela `(fil,col)` del nuevo objeto tendrá como altura el valor máximo entre las alturas de dicha parcela en los objetos involucrados.

Para implementar estos métodos, puede incorporar los métodos adicionales que considere oportunos. Si lo hace, justifique si deben ser públicos o privados. Además, considere que dispone de la implementación ya terminada de las clases `Punto2D` y `SecuenciaPuntos`, mostradas en la tabla 2.

Punto2D
- int x - int y
+ Punto2D() + Punto2D(int abscisa, int ordenada) + int Abscisa() + int Ordenada() + double Distancia(Punto2D otro)

SecuenciaPuntos
- static const int TAMANIO = 100 - Punto2D vector_privado[TAMANIO] - int total_utilizados
+ SecuenciaPuntos() + int Capacidad() + int TotalUtilizados() + void Aniade(Punto2D pto) + Punto2D Elemento(int indice)

Table 2: Clases disponibles y métodos que **NO hace falta implementar**.

Suponga que en `main` dispone de objetos `r1` y `r2` ya creados de la clase `Relieve`. Escriba el código necesario para:

1. (0.75) Mostrar las coordenadas y la altura del pico más alto de `r1`.
2. (0.75) Calcular la longitud de cable requerida para unir los picos de `r1` (orden: el dado por `ObtenerPicos`). Debe tener en cuenta la altura de los picos.
3. (0.5) Mostrar las coordenadas de los picos del relieve que se obtendría al fusionar los objetos `r1` y `r2`.

◁ Ejercicio 3 ▷ Rima asonante [3 puntos]

Dispone de la implementación de la clase `SecuenciaCaracteres` mostrada en la tabla 3. Sobre dicha clase, implemente un método que compruebe si una secuencia rima de forma asonante con otra secuencia con grado `k`. Esto significa que las últimas `k` vocales de ambas secuencias deberán coincidir. El método devolverá `true` si las dos secuencias riman de forma asonante, y `false` en caso contrario.

Implemente todos los métodos auxiliares que estime oportuno. No hace falta construir el programa principal, pero incluya al menos la(s) línea(s) en la(s) que se realizaría la llamada al método pedido.

SecuenciaCaracteres
- static const int TAMANIO = 100 - char vector_privado[TAMANIO] - int total_utilizados
+ SecuenciaCaracteres() + int Capacidad() + int TotalUtilizados() + void Aniade(char nuevo) + char Elemento(int indice)

Table 3: Clase disponible y métodos que **NO hace falta implementar**.