Curso de nivelación de algoritmos

Examen final

Diciembre de 2020

Ejercicio 1 (15 pts)**.** Dado un entero n mayor a 0, se define al n-ésimo término de la sucesión de Padovan como:

$$P(n) = \begin{cases} 1 & \text{si } n = 1,\ 2 \text{ o } 3 \\ P(n-2) + P(n-3) & \text{si } n > 3 \end{cases}$$

Los primeros términos de la sucesión son 1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 12, 16, ...

Definir una función ***recursiva*** `padovan(n)` en Python que reciba un número entero n mayor a 0 y devuelva el n-ésimo término de la sucesión de Padovan. Ejemplo de uso de la función:

```
# Calcula el 10º número de Padovan.
res_padovan = padovan(10)

# Imprime un 9 por pantalla.
print(res_padovan)
```

Ejercicio 2 (15 pts)**.** Dados los siguientes dos programas:

```
def f1(L):
    i = a = e = 0
    while i < len(L):
        if L[i] == 'a':
            a = a + 1
        elif L[i] == 'e':
            e = e + 1
        i = i + 1
    return a == e
```

```
def f2(L):
    r = 0
    j = len(L)-1
    while j >= 0:
        if L[j] == 'a':
            r = r + 1
        j = j - 1
    while j < len(L)-1:
        if L[j+1] == 'e':
            r = r - 1
        j = j + 1
    return r == 0
```

a) Determinar qué computa cada función. Justificar brevemente.

b) Determinar para cada función cuál es el orden de complejidad algorítmica en el peor caso en función del tamaño de la lista L. Justificar brevemente. *Ayuda*: cada función se corresponde con alguno de los siguientes órdenes de complejidad, donde $n = len(L)$

$$O(1),\ O(n),\ O(log(n)),\ O(n^2),\ O(2^n),\ O(n * log(n)),\ O(n^n),\ O(n^3).$$

Ejercicio 3 (10 pts)**.** Determinar cuál es el valor de las variables a, b, c, i luego de ejecutar el siguiente programa. Expresar los resultados en función de k cuando sea necesario, donde k es una variable que almacena un número entero y que fue definida con anterioridad en el programa.

```
a, b, c = k, 2*k, 3*k
i = 5
while i <= 9:
   c = a
   a = b - a
   b = c
   i = i + 2
```

Ejercicio 4 (10 pts)**.** Indique *Verdadero* o *Falso* para cada una de las siguientes afirmaciones acerca de la recursión. Explique brevemente cada una de sus respuestas.

1. Permite resolver problemas que con el `while` no es posible resolver.
2. Los algoritmos recursivos siempre terminan.
3. El backtracking hace uso de la recursión.
4. No es necesario que un algoritmo recursvio tenga un caso base.
5. Los algoritmos recursivos tienen menor complejidad algorítmica que los iterativos.

Ejercicio 5 (25 pts)**.** Definir una función `primeros_primos(n)` en Python que reciba un número entero n mayor a 0 y devuelva una lista con los primeros n números primos. Por ejemplo:

`primeros_primos(1)` devuelve la lista `[2]`
`primeros_primos(2)` devuelve la lista `[2,3]`
`primeros_primos(9)` devuelve la lista `[2, 3, 5, 7, 11, 13, 17, 19, 23]`

Explicar conceptualmente qué hace cada parte del programa (no explicar línea por línea el programa). Sugerencia: definir las funciones *es_primo(n)* y *enesimo_primo(n)*.

Ejercicio 6 (15 pts)**.** Sea la siguiente función, cuya entrada es una lista de números:

```
def func(L):
   i = r = 0
   n = len(L)
   while i < n:
      j = i
      i = 1
      while i <= L[j] and i*i != L[j]:
         i = i + 1
      if not(i > L[j]):
         r = r + 1
      i = j + 1
   return r == n
```

a) Renombrar a `func` con un nombre que sea más descriptivo de lo que computa (máximo 4 palabras). Por ejemplo, si `func` ordenase la lista L, un mejor nombre podría ser `ordenar_lista` o simplemente `ordenar`. Argumentar la elección del nuevo nombre explicando qué hace la función (no explicar línea por línea el programa). Si le resulta útil, puede acompañar la explicación con ejemplos de listas y el resultado de aplicar la función.

b) ¿Qué devuelve `func` si la lista contiene números negativos? ¿Por qué?

Ejercicio 7 (10 pts)**.** Indique *Verdadero* o *Falso* para cada una de las siguientes afirmaciones acerca del algoritmo *Selection Sort*. Explique brevemente cada una de sus respuestas.

1. Su complejidad algorítmica en peor caso es mayor que en el *Insertion Sort*.
2. Si la lista ya está ordenada termina más rápido que si está al revés.
3. Su estrategia es buscar el siguiente mínimo y ponerlo en la posición definitiva.
4. Su complejidad algorítmica en peor caso es mayor que en el *Merge Sort*.
5. Su estrategia es ordenar recursivamente todos los elementos menos el primero y luego agregar este en la posición definitiva.