Prácticas de Programación
PEC4 - 20222

Fecha límite de entrega: **21 / 05 / 2023**

Formato y fecha de entrega

La PEC debe entregarse antes del día **21 de mayo de 2023** a las 23:59.

Es necesario entregar un fichero en formato **ZIP** que contenga:

- Un documento en formato **pdf** con las respuestas a los **ejercicios**. El documento tiene que indicar en su primera página el **nombre y apellidos** del estudiante que hace la entrega.

La entrega debe hacerse en el apartado de entregas de AC del aula de teoría antes de la fecha de entrega. **Sólo la última entrega** dentro del periodo estipulado será evaluada.

Objetivos

- Adquirir los conceptos teóricos explicados sobre los métodos de búsqueda.
- Interpretar los algoritmos de búsqueda, pudiendo simular su funcionamiento dada una entrada.
- Implementar cualquier método de búsqueda en un algoritmo y evaluar su rendimiento.
- Adquirir los conceptos teóricos explicados sobre las técnicas de análisis de algoritmos.
- Analizar la complejidad de una función, calculando su función de tiempo de ejecución y expresar esta complejidad con notación asintótica.

Criterios de corrección:

Cada ejercicio tiene asociada su puntuación sobre el total de la actividad. Se valorará tanto que las respuestas sean correctas como que también sean completas.

- No seguir el **formato de entrega**, tanto por lo que se refiere al **tipo y nombre de los ficheros** como al contenido solicitado, comportará una **penalización importante** o la calificación con una **D de la actividad**.
- En los ejercicios en que se pide lenguaje algorítmico, se debe respetar el **formato.**
- En el caso de ejercicios en lenguaje C, estos **deben compilar para ser evaluados**. Si compilan, se valorará:
    - Que **funcionen** tal como se describe en el enunciado.
    - Que se respeten los **criterios de estilo y** que el código esté **debidamente comentado**.
    - Que las **estructuras** utilizadas sean las correctas.

Aviso

Aprovechamos para recordar que **está totalmente prohibido copiar en las PECs** de la asignatura. Se entiende que puede haber un trabajo o comunicación entre los estudiantes durante la realización de la actividad, pero la entrega de esta debe que ser individual y diferenciada del resto. Las entregas serán analizadas con **herramientas de detección de plagio.**

Así pues, las entregas que contengan alguna parte idéntica respecto a entregas de otros estudiantes serán consideradas copias y todos los implicados (sin que sea relevante el vínculo existente entre ellos) suspenderán la actividad entregada.

Guía citación:
https://biblioteca.uoc.edu/es/contenidos/Como-citar/index.html

Monográfico sobre plagio:
http://biblioteca.uoc.edu/es/biblioguias/biblioguia/Plagio-academico/

Observaciones

Esta PEC continúa el proyecto que se desarrolla durante las distintas actividades del semestre.

En este documento se utilizan los siguientes símbolos para hacer referencia a los bloques de diseño y programación:

 Indica que el código mostrado es en **lenguaje algorítmico**.

 Indica que el código mostrado es en **lenguaje C**.

 Muestra la **ejecución** de un programa en **lenguaje C**.

Enunciado

En la PAC1 se introdujo el concepto de donación (tDonation) que una persona hace a un proyecto de una ONG con una determinada cantidad y en una fecha concreta. En esta PEC entraremos algo más en detalle en la gestión de estas donaciones y trabajaremos algunos conceptos relacionados con el cálculo de complejidad algorítmica y los algoritmos de búsqueda.

Ejercicio 1: Conceptos sobre búsqueda y complejidad [20%]

En la PEC1 implementamos los siguientes tipos de datos para guardar las donaciones:

```
const
      MAX_DONATIONS: integer = 120;
end const

type
      tDonation = record
            date: tDate;
            personDocument: string;
            ngo: string;
            projectCode: string;
            amount: real;
      end record

      tDonationData = record
            elems: vector [MAX_DONATIONS] of tDonation;
            count: integer;
      end record

end type
```

Considerando esta implementación, y que los registros de donaciones están **ordenados utilizando el documento de la persona**, responde a las siguientes preguntas razonando tu respuesta:

a) [5%] Suponiendo que la lista de donaciones ***tDonationData*** tiene 6 donaciones de 6 personas diferentes, ¿cuántas iteraciones harán falta como máximo para encontrar una donación si hacemos uso de la búsqueda secuencial? ¿y haciendo uso de la búsqueda binaria? Pon un ejemplo de los índices que habría que consultar para buscar una donación que no está en la lista.

Haciendo uso de la búsqueda secuencial, como máximo tendremos que recorrer las 6 donaciones, es decir, haremos 6 iteraciones. Por otro lado, si usamos la búsqueda binaria, como máximo tendremos que hacer 3 iteraciones. Ejemplos de los índices que habría que consultar con la búsqueda binaria son:

- 3, 1 y 2
- 3, 5 y 4
- 3, 5 y 6

b) [5%] Si la lista de donaciones ***tDonationData*** estuviera implementada con punteros, donde cada donación fuera un nodo enlazado con el siguiente con un puntero "next", ¿qué método de búsqueda sería más eficiente para este caso particular, el secuencial o el binario?

En las listas implementadas con punteros no puede utilizarse la búsqueda binaria puesto que no se puede acceder de manera directa a los nodos de la lista y hay que hacer un recorrido con los punteros "next".

c) [10%] Calcula la complejidad de las siguientes funciones de tiempo de ejecución utilizando la notación asintótica y ordénalas de más eficiente a menos eficiente:

1) $T(n) = 16 - n + 2^{n+1}$
2) $T(n) = n + n^2 + 1000$
3) $T(n) = 999$
4) $T(n) = n - 2$
5) $T(n) = n \cdot \log_2(n+2)$
6) $T(n) = 256^2 + \log(2)$

Las complejidades computacionales ordenadas son:

3)	$O(1)$	constante
6)	$O(1)$	constante
4)	$O(n)$	lineal
5)	$O(n \log(n))$	casi-lineal
2)	$O(n^2)$	cuadrática
1)	$O(2^n)$	exponencial

Ejercicio 2: Algoritmos de búsqueda [30%]

Hemos ordenado las donaciones (***tDonationData***) por el documento de la persona para facilitar la búsqueda por persona. Sin embargo, en este ejercicio, para simplificar, en lugar de trabajar con el vector de elementos de tipos ***tDonation***, trabajaremos con un vector de identificadores numéricos (enteros).

Así pues, dado el siguiente vector **ordenado**:

1	2	3	4	5	6	7	8	9	10	11	12
5	7	11	17	25	34	43	59	63	71	88	91

a) [5%] Aplica el algoritmo de **búsqueda lineal** para encontrar el elemento 59.

```
El elemento 59 se encuentra en la posición 8 del
vector:

índice 1:  5 ≠ 59
índice 2:  7 ≠ 59
índice 3:  11 ≠ 59
índice 4:  17 ≠ 59
índice 5:  25 ≠ 59
índice 6:  34 ≠ 59
índice 7:  43 ≠ 59
índice 8:  59 = 59
```

b) [5%] Aplica el algoritmo de **búsqueda lineal** para encontrar el elemento 33.

```
El elemento 33 no se encuentra en el vector:

índice 1:  5 ≠ 33
índice 2:  7 ≠ 33
índice 3:  11 ≠ 33
índice 4:  17 ≠ 33
índice 5:  25 ≠ 33
índice 6:  34 ≠ 33
índice 7:  43 ≠ 33
índice 8:  59 ≠ 33
índice 9:  63 ≠ 33
```

```
índice 10: 71 ≠ 33
índice 11: 88 ≠ 33
índice 12: 91 ≠ 33

Dado que el vector se encuentra ordenado, también es
correcta la solución que detiene la búsqueda en el
índice 6, puesto que a partir de este índice todos los
elementos serán mayores o iguales que 34, y, por lo
tanto, el 33 no estará entre ellos.
```

c) [10%] Aplica el algoritmo de **búsqueda binaria** para encontrar el elemento 70.

```
El elemento 70 no se encuentra en el vector:

I=1 D=12          (1+12)/2 = 13/2 = 6   índice 6:   34 <
70
I=7 D=12          (7+12)/2 = 19/2 = 9   índice 9:   63 <
70
I=10 D=12     (10+12)/2 = 22/2 = 11 índice 11: 88 > 70
I=10 D=10     (10+10)/2 = 20/2 = 10 índice 10: 71 > 70
I=10 D=9           No encontrado
```

d) [10%] Aplica el algoritmo de **búsqueda binaria** para encontrar el elemento 25.

```
El elemento 25 se encuentra en la posición 5 del
vector:

I=1 D=12          (1+12)/2 = 13/2 = 6    índice 6:   34
> 25
I=1 D=5       (1+5)/2 = 6/2 = 3      índice 3:  11 < 25
I=4 D=5       (4+5)/2 = 9/2 = 4      índice 4:  17 < 25
I=5 D=5       (5+5)/2 = 10/2 = 5     índice 5:  25 = 25
```

Ejercicio 3: Uso de los algoritmos de búsqueda [20%]

A partir de la siguiente definición de la lista de proyectos de las ONG:

```
const
      MAX_PROJECTS: integer = 100;
end const

type
      tProject = record
           code: string;
           ngoCode : string;
           budget: real;
      end record


      tProjectData = record
            elems: vector [MAX_PROJECTS] of tProject;
            count: integer;
      end record

end type
```

Asumiendo que los elementos de la lista ***tProjectData*** se encuentran ordenados por el campo *code* del proyecto en orden alfabético, se pide implementar en lenguaje algorítmico la función:

```
function checkProjectBudget (projects: tProjectData, projectCode:
string, budget: real) : boolean
```

que, dada la lista de proyectos (*projects*), el código de un proyecto (*projectCode*) y un presupuesto (*budget*), nos devuelva ***true*** si el presupuesto asignado a dicho proyecto es igual o mayor al presupuesto pasado por parámetro. En caso contrario o en caso de no encontrar dicho proyecto en la lista, la función tiene que devolver ***false***. Para la implementación hay que seguir el método de **búsqueda binaria**.

```
function checkProjectBudget (projects: tProjectData, projectCode:
string, budget: real) : boolean

var
   left, right, middle : integer;
   found, res : boolean;
end var

   { Initialize variables }
   left := 1;
   right := projects.count;
```

```
    found := false;
    res := false;

    { Loop using binary search }
    while (left ≤ right) and not found do
       middle := (left + right) div 2;

       if projects.elems[middle].code = projectCode then
          found := true;
          if projects.elems[middle].budget ≥ budget then
             res := true
          end if
       else
         if projects.elems[middle].code < projectCode then
            left := middle + 1;
         else
            right := middle -1;
         end if
       end if
    end while

    return res;

end function
```

Ejercicio 4: Análisis de algoritmos [30%]

Responde a las preguntas siguientes:

a) **[15%]** Dada la siguiente definición del tiempo de ejecución, calcula la función T(n) sin que aparezca ninguna definición recursiva y explica el proceso que has seguido para obtener el resultado.

$$\begin{cases} T(n) = a, & \text{si } n=0 \\ T(n) = b + n + T(n-1), & \text{si } n>0 \end{cases}$$

El método para resolver este tipo de ecuaciones consiste en aplicar la definición recursiva de forma repetida un total de *i* veces hasta que vemos qué forma adopta la expresión resultante:

T(n) =
= **b+n+T(n-1)** = b+n+(b+(n-1)+T(n-1-1)) =
= **2b+2n-1+T(n-2)** = 2b+2n-1+(b+(n-2)+T(n-2-1)) =
= **3b+3n-3+T(n-3)** = 3b+3n-3+(b+(n-3)+T(n-3-1)) =
= **4b+4n-6+T(n-4)** = … =

$$ib + in - \sum_{j=0}^{i-1} j + T(n-i)$$

Averiguamos qué valor tiene que tomar *i* para poder aplicar el caso base de la definición de T(n):

n-i = 0
n=i

A continuación, completamos el cálculo aprovechando que la expresión T(n-i) cuando i = n se transforma en T(0), que según la definición del enunciado es a:

$$T(n) = nb + nn - \sum_{j=0}^{n-1} j + T(n-n) = bn + n^2 - \frac{n(n-1)}{2} + T(0) =$$

$$= n^2 + bn - \frac{n^2-n}{2} + a = \frac{1}{2}n^2 + (b+\frac{1}{2})n + a$$

b) [15%] Calcula la complejidad computacional de la función **passed_matrix_count** y explica el proceso que has seguido para obtener el resultado:

```
function passed_matrix_count (marks: vector[N][N] of
real): integer
   var
     mark: real;
     passed: integer;
     i, j: integer;
   end var
   mark := 0.0;                          (1)
   passed := 0;                          (2)
   i := 1;                               (3)
   j := 1;                               (4)
   while i <= N do                       (5)
     while j <= N do                     (6)
       mark := marks[i][j];              (7)
       if mark >= 5.0 then               (8)
         passed := passed + 1;                (9)
       end if                            (10)
       j := j + 1;                       (11)
     end while                           (12)
     i := i + 1;                         (13)
     j := 1;                             (14)
   end while                             (15)
   return passed;                        (16)
end function
```

La función **passed_matrix_count** empieza con cuatro asignaciones (líneas 1-4) que son operaciones elementales con un tiempo constante que se pueden agrupar en k_1.

Después hay un bucle externo (líneas 5-15). El tiempo total del bucle externo es la multiplicación del coste de una iteración por el número de iteraciones, que, en este caso, es igual al número de filas de la matriz (N). El coste de una iteración es la suma de una serie de operaciones elementales y el coste del bucle interno (líneas 6-12). El coste de las operaciones constantes las agrupamos en el valor constante k_2, concretamente el coste de las siguientes operaciones: comparación (línea 5), asignaciones

(líneas 13 y 14) e incremento de la variable contador *i* del bucle (línea 13).

El tiempo del bucle interno es la multiplicación del coste de una iteración por el número de iteraciones, que en este caso, es igual al número de columnas de la matriz (N). El siguiente paso es calcular el coste de una iteración del bucle interno, que es la suma de una secuencia de operaciones elementales con coste constante: las comparaciones (líneas 6 y 8), las asignaciones (líneas 7, 9 y 11), el incremento de la variable contador *j* del bucle (línea 11), los accesos a la matriz (línea 7) y la operación aritmética de incremento en una unidad (línea 9). El resultado de esta suma es el coste de una iteración y lo agrupamos en el valor constante k_3. Así pues, el coste total del bucle interno es k_3*N.

Ahora podemos completar el coste del bucle externo que recorre la matriz y queda como: $(k_2 + k_3*N)*N = k_2*N + k_3*N^2$.

Por lo tanto, la función del tiempo de ejecución de **passed_matrix_count** es:

$$T(N) = k_1 + k_2*N + k_3*N^2$$

Por lo tanto, la complejidad es **cuadrática**: $O(N^2)$.