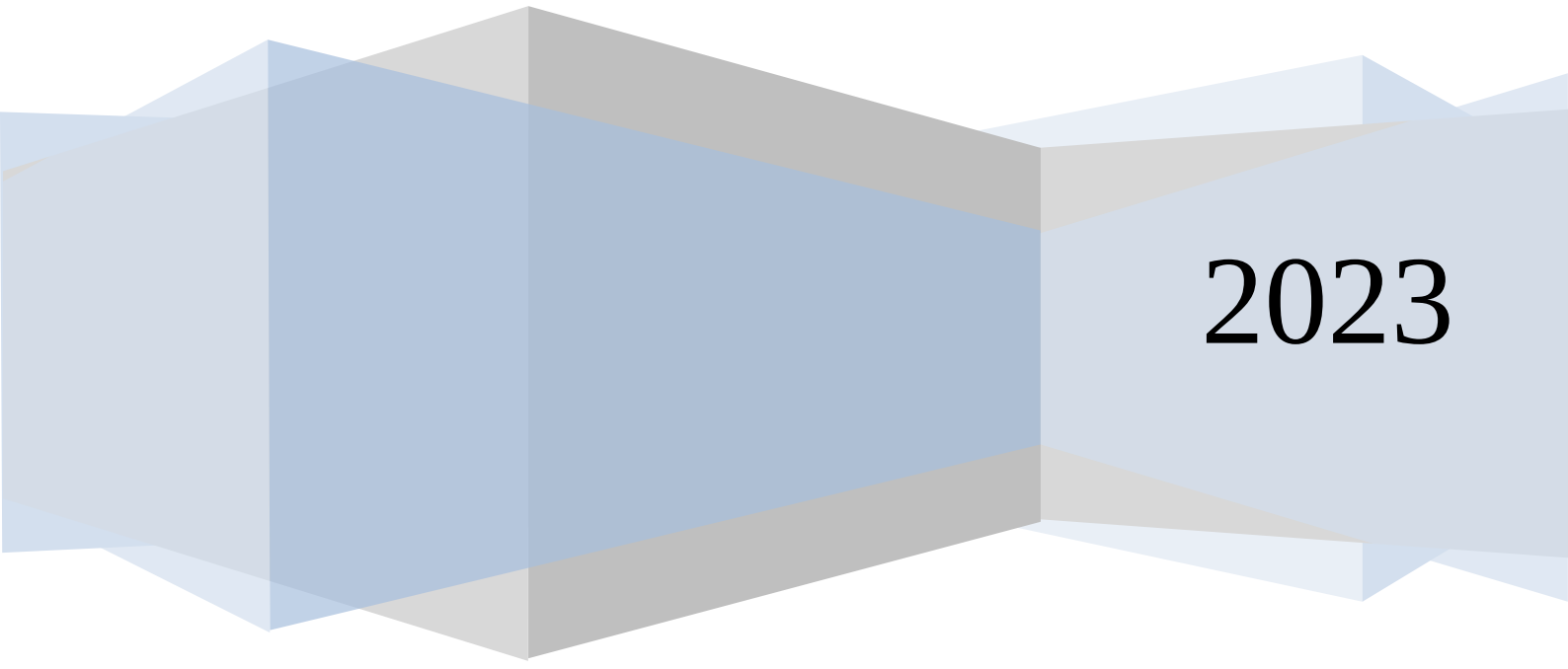
Programación II

Practica 5:

Complejidad

David Piñuel Bosque



2023

Índice

1. **Ejercicio 1 complejidad.**

```
function Recursiva(n: natural)
    begin
    if (n = 0)
          return 1;
       else
            return 1 + Recursiva(n - 1);
    end if;
end function;
```

Cuando n = 0, la función se detiene y retorna 1. Esto cuenta como una operación.

Cuando n es mayor que 0, la función se llama a sí misma con un argumento decrementado en 1 (Recursiva(n - 1)) y se realiza una suma (1 + Recursiva(n - 1)). Estas dos operaciones se realizan en cada llamada recursiva.

Podemos definir una función T(n)que representa el número total de operaciones para un valor de entrada n.

Para el caso base, cuando n = 0, la función realiza 1 operación, por lo que tenemos T(0) = 1.

Para n > 0, podemos expresar el número total de operaciones como:

T(n) = 1 + T(n - 1)

Esto significa que el número total de operaciones es igual a 1 (la suma) más el número total de operaciones para n - 1.

Expandiendo la formulación, tenemos:

T(n) = 1 + (1 + T(n - 2))
= 1 + 1 + T(n - 2)
= 1 + 1 + 1 + T(n - 3)
= ...
= 1 + 1 + 1 + ... + 1 + T(0)

La cantidad de veces que se suma 1 es n, ya que en cada llamada recursiva n se decrementa en 1 hasta llegar a 0.

Simplificando la formulación, obtenemos:

T(n) = n * 1 + T(0)
= n + T(0)

La parte T(0) representa el número de operaciones cuando n = 0, que es 1.

Por lo tanto, el número total de operaciones para un valor de entrada n es:

T(n) = n + 1

En términos de orden de complejidad, nos interesa el término dominante a medida que n crece. En este caso, el término dominante es n.

Por lo tanto, el orden de complejidad de la función Recursiva es O(n), lo que significa que el número de operaciones crece linealmente con el valor de entrada n.

2. **Ejercicio 2 Complejidad.**

```
void Floyd(int C[][], int A[][], int P[][], int nNodos) {
  int i, j, k;
  for (i = 0; i < nNodos; i++) {
     for (j = 0; j < nNodos; j++) {
        A[i][j] = C[i][j];
        P[i][j] = -1;
     }
  }
  for (k = 0; k < nNodos; k++) {
     for (i = 0; i < nNodos; i++) {
        for (j = 0; j < nNodos; j++) {
           if (A[i][k] + A[k][j] < A[i][j]) {
              A[i][j] = A[i][k] + A[k][j];
              P[i][j] = k;
           }
        }
     }
  }
}
```

Vamos a analizar el número total de operaciones en función del número de nodos nNodos en el grafo.

El primer bucle para se ejecuta nNodos veces. Dentro de este bucle, hay dos operaciones de protección en cada iteración: A[i][j] = C[i][j]y P[i][j] = -1.

Cada una de estas operaciones tiene un tiempo de ejecución constante, por lo que el primer bucle tiene un tiempo de ejecución de O(nNodos^2) debido a las dos dimensiones de la matriz.

Luego, hay un segundo bucle para que también se ejecute nNodos veces.

Dentro de este bucle, hay un tercer bucle que también se ejecuta nNodosa veces.

Dentro del tercer bucle for, se realiza una comparación (A[i][k] + A[k][j] < A[i][j]) y una operación de preselección (A[i][j] = A[i][k] + A[k][j]) en el condicional if. Estas operaciones tienen un tiempo de ejecución constante.

Por lo tanto, el segundo y tercer bucle for, en total, tienen un tiempo de ejecución de O(nNodos^3) debido a las tres dimensiones de las matrices involucradas.

En general, el tiempo de ejecución de la función Floyd está dominado por el segundo y tercer bucle for, lo que resulta en un orden de complejidad de O(nNodos^3).

3. Aclaraciones y comentarios.

En el ejercicio 1 he realizado la complejidad de la función recursiva y en el ejercicio 2 he realizado la complejidad de la función Floyd. He dedicido realizarla de esta manera redactando paso a paso a detalle su complejidad que van teniendo cada función.