Laboratorio - 5

Alumno: Jackson Fernando Merma Portocarrero

Escuela Profesional: Ingeniería de Sistemas

CUI: 20202143

Correo: jmermap@unsa.edu.pe

Generalidades

- La explicación de estos problemas están también en el código.
- Estos problemas tienen entrada de datos (generalmente solo para ‘n’).
- Los 12 problemas, se encuentran en una subcarpeta ‘12_ejercicios’.
- Los otros 2 problemas se encuentran a la altura de este informe (directorio ‘clase_05’).

Parte 1: 2 problemas faltantes

- Problema de coincidencia de una cadena en otra: **filename** → *coincidencia_palabras.cpp*

```
# 4. What is the time complexity in terms of O()?
"fgh" "abcdefghi"
def find_needle(needle, haystack)
    needle_index = 0
    haystack_index = 0

    while haystack_index < haystack.length
        if needle[needle_index] == haystack[haystack_index]
            found_needle = true
            while needle_index < needle.length
                if needle[needle_index] != haystack[haystack_index + needle_index]
                    found_needle = false
                break
            end
            needle_index += 1
            end
        return true if found_needle
        needle_index = 0
        end
        haystack_index += 1
    end
    return false
end
```

Explanation:

```
/*
 * Complejidad --> O(n*m)
 * --> n : str1.length()
 * --> m : str2.length()
 *
 *
 * Explicacion: En el primer 'while' se recorre a 'm'. Luego, si se encuentra una
 *              coincidencia
 *              en la primera letra de str1 y en la posicion i-esima de m, entonces
 *              recorre toda la longitud de str1 en str2.
 *
 *              Explicacion 1:
 *
 *                En otras palabras, en el peor caso str1, seria una cadena de n-1
 *                caracteres iguales y un caracter final diferente, ejemplo:
 *                -  aaaaax
 *                Por otro lado str2, seria una cadena de m caracteres iguales al
 *                primer caracter de str1, ejemplo :
 *                - aaaaaaaaaaaaaaaaaaaaaa
 *
 *                Entonces, en general estos casos, darian como resultado n*m
 *                comparaciones aproximadamente (peor caso).
 *
 *
 *              Explicacion 2: Usando una matriz de n*m
 *
 *                Si se representara a 0 y 1 con "coincide" y "no coincide"
 *                (caracter) respectivamente.
 *                Entonces:
 *
 *                -> pasada 1: Primer caso
 *
 *                   1  2  3  4  5  6  7  8  ...  m
 *                1  1
 *                2     1
 *                3        1
 *                4           1
 *               ...
 *                n              0                  -> el primer caso no coincide
 *
 *                -> pasada k-esima: Ultimo caso
 *
 *                   1  2  3  4  5  6  7  8  ...  m
 *                1  1  1  1  1  1  1  1  1
 *                2     1  1  1  1  1  1  1
 *                3        1  1  1  1  1  1
 *                4           1  1  1  1  1
 *               ...
 *                n              0  0  0  0  ...  1 ->todos fallan, excepto el ultimo
 *
 *                Igual que en el explicacion 1, tendria que comparar en general, n*m
 *                posiciones y al
 *                final encontrar o no encontrar la coincidencia
 *
 *
 */
```

Input:

```
hola
hola me llamo Jackson
```

Output:

```
'hola' in 'hola me llamo Jackson'!
```

- Problema de encontrar una posición especial: **filename** → *pick_resume.cpp*

```
# 5. What is the time complexity in terms of O()?
# resumes is an array
def pick_resume(resumes):
    eliminate = "top"

    while resumes.length > 1:
        if eliminate == "top":
            resumes = resumes[resumes.length / 2, resumes.length - 1]
            eliminate = "bottom"
        elif eliminate == "bottom":
            resumes = resumes[0, resumes.length / 2]
            eliminate = "top"
        end
    end
    return resumes[0]
end
```

```
Explanation:
/*
 * Complejidad --> O(log2(n))
 *
 * Explicacion: En este caso, solamente se juega con los limites (a y b), donde
 *              segun el valor de 'top' cambian su valor, pero estos disminuyen a
 *              la mitad de la longitud que va quedando en el arreglo, entonces
 *              esto quiere decir que van visitando 'mitades' del arreglo.
 *
 *              Por ende, el resultado para encontrar a esa mitad unica(cuando la
 *              longitud es 1), se encuentra luego de log2(n) busquedas.
 *
 */
```

```
Input:
8
1 2 3 4 5 6 7 8

Output:
answer:6
```

Parte 2: 12 problemas

- Question 1

```
// Q1: What is the time complexity of
for (i = 0; i < n; i++) {
    statement;
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> lineal --> O(n)
 *
 * Explicacion: Recorre un bucle 'n' veces --> rango: [0 a n-1] = [1 a n] = n
 *              veces
 */
```

```
Input:
4

Output:
hola!
hola!
hola!
hola!
```

- Question 2

```
for (i = n; i > 0; i--) {
    statement;
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> lineal --> O(n)
 *
 * Explicacion: Recorre un bucle 'n' veces --> rango: [n a 1] = [1 a n] = n veces
 */
```

```
Input:
4

Output:
hola!
hola!
hola!
hola!
```

- Question 3

```
for (i = 0; i < n; i=i+5) {
    statement;
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> lineal --> O(n)
 *
 * Explicacion: Recorre un bucle 'n/5' veces aproximadamente
 *              --> rango: [0 a n-1] de 5 en 5 = [1 a n]/5 = n/5 veces
 */
```

```
Input:
10

Output:
hola!
hola!
```

- Question 4

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        statement;
    }
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> cuadratico --> O(n^2)
 *
 * Explicacion: Recorre dos bucles 'n^2' veces
 *              --> rango: [0 a n-1]*[0 a n-1] = [1 a n]*[1 a n] = n*n = n^2
 *                  veces
 */
```

Input:
3

Output:
hola!
hola!
hola!
hola!
hola!
hola!
hola!
hola!
hola!

- Question 5

```
for (i = 0; i < n; i++) {
    for (j = 0; j < i; j++) {
        statement;
    }
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> cuadratico --> O(n^2)
 *
 * Explicacion: Recorre dos bucles 'n*(n^2 + 1)/2' veces
 *                           --> rango: [0 a n-1]*[0 a n-1 en la mitad de veces] =
 *
 *                               1   2    3   4   5   ...  n
 *                           1   0
 *                           2   1   0
 *                           3   1   1    0
 *                           4   1   1    1   0
 *                           5   1   1    1   1   0
 *                          ...
 *                           n   1   1    1   1   1   ...  0
 *
 *                           => recorre n*(n^2 + 1)/2' veces
 *
 */
```

Input:
5

```
Output:
0
1 0
1 1 0
1 1 1 0
1 1 1 1 0
```

- Question 6

```
p = 0
for (i = 1; p <= n; i++) {
    p = p + i;
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> raiz cuadrada --> sqrt(n)
 *
 * Explicacion: Este depende del crecimiento de 'p', el cual crece -> m*(m+1)/2
 *              veces, entonces si debe ser menor o igual que 'n', este proceso
 *              se corta cuando:
 *              [crecimiento triangular] O(m*(m+1)/2) > n  ==> m^2 > n ==> m >
 *              sqrt(n) veces
 */
```

```
Input:
25

Output:
hola!
hola!
hola!
hola!
hola!
hola!
hola!
```

- Question 7

```
for (i = 1; i < n; i = i*2) {
    statement;
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> logaritmico --> log2(n)
 * Explicacion: Recorre un bucle log2(n) veces
 *              --> dado que el crecimiento es n, n/2, n/4, n/8, ... n/2^k
 *              --> dado que el crecimiento es 1, 2, 4, 8, 16, ... 2^k
 *              --> luego, si n es $n se recorre $m
 *                              $n    -     $m
 *                               2    -     1
 *                               4    -     2
 *                               8    -     3
 *                               16   -     4
 *
 *                               ...  -     ...
 *
 *                               2^k  -     log2(n)
 *               --> entonces, si debe ser mayor o igual que 1, 'i' recorreria
 *                   log2(n) veces
 */
```

```
Input:
8

Output:
log2(8)=3
```

- Question 8

```
for (i = n; i >= 1; i = i/2) {
    statement;
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> algoritmico --> O(log2(n))
 *
 * Explicacion: Recorre un bucle log2(n) veces
 *              --> dado que el crecimiento es n, n/2, n/4, n/8, ... n/2^k
 *              --> luego, si n es $n se recorre $m
 *                                  $n    -     $m
 *                                   2    -     1
 *                                   4    -     2
 *                                   8    -     3
 *                                   16   -     4
 *
 *                                   ...  -     ...
 *
 *                                   2^k  -     log2(n)
 *                --> entonces, si debe ser mayor o igual que 1, 'i' recorreria
 *                    log2(n) veces
 */
```

```
Input:
16

Output:
log2(16)=4
```

- Question 9

```
for (i = 0; i * i < n; i++) {
    statement;
}
```

```
Explanation:
/*
 * Complejidad del algoritmo --> raiz cuadrada --> O(sqrt(n))
 *
 * Explicacion: Recorre un bucle sqrt(n) veces
 *              --> dado que el crecimiento es 0, 1, 4, 9, 16, i^k
 *              --> luego, si n es $n se recorre $m, dando resultado a $ans
 *                           $n    -     $m                    =     $ans
 *                            0    -     0                     =       0
 *                            1    -     0,1                   =       1
 *                            4    -     0,1,2                 =       2
 *                            9    -     0,1,2,3               =       3
 *                            16   -     0,1,2,3,4             =       4
 *
 *                           ...   -       ...                 =      ...
 *
 *                            n    -     0,1,2,3,... sqrt(n)   =    sqrt(n)
 *                --> entonces, recorreria sqrt(n) veces
```

```
 */
```

Input:
49

Output:
sqrt(49)=7

- Question 10

```
for (i = 0; i < n; i++) {
    statement;
}

for (j = 0; j < n; j++) {
    statement;
}
```

Explanation:

```
/*
 * Complejidad del algoritmo --> lineal --> O(n)
 *
 * Explicacion: Recorre un bucle '2n' veces
 *              --> esto ya que repite otra vez un bucle ya definido de
 *                  [0 a n-1] -> [1 a n]
 *              --> Entonces, el coste será de 2*n veces
 */
```

Input:
4

Output:
2*4=8

- Question 11

```
p = 0
for (i = 1; i < n; i = i * 2) {
    p++;
}

for (j = 1; j < p; j = j * 2) {
    statement;
}
```

Explanation:

```
/*
 * Complejidad del algoritmo --> O(log2(log2(n)))
 *
 * Explicacion: Recorre 2 bucles donde el primero depende de 'n' y el segundo de
 *              'p'
 *              --> Si en el primero hay un tiempo logaritmico, eso quiere decir
 *                  que:
 *                                  p = log2(n)
 *              --> Luego, en el segundo tambien hay un tiempo logaritmico, para
 *                  lo cual:
 *                                  m = log2(p)
 *              --> Entonces, el coste será log2(log2(n)) veces
 */
```

Input:
256

Output:
log2(log2(256))=3

- Question 12

```
for (i = 0; i < n; i++) {
    for (j = 1; j < n; j = j * 2) {
        statement;
    }
}
```

Explanation:

```
/*
 * Complejidad del algoritmo --> O(n*(log2(n)))
 *
 * Explicacion: Recorre dos bucles (1 anidado)
 *              --> El primero solo depende de 'n', entonces el coste es 'n'
 *              --> Luego, el segundo tambien depende de 'n', pero el
 *                  crecimiento es de log2(n)
 *
 *              --> Finalmente, se puede decir que el coste es la multiplicacion
 *                  => n*log2(n)
 */
```

Input:
4

Output:
4*log2(4)=8