ESTRUCTURA DE DATOS 1
Código ST0245

Laboratorio Nro. 1
Recursión

Brahyan Esteban Ríos Soto
Universidad Eafit
Medellín, Colombia
berioss@eafit.edu.co

Nombre completo de integrante 2
Universidad Eafit
Medellín, Colombia
Correointegrante2@eafit.edu.co

2) Simulacro de Maratón de Programación sin documentación HTML, en GitHub, en la carpeta ejercicioEnLinea.

2.1 Recursión 1:

2.1.1
```
public int factorial(int n) {
        if (n == 1)
              return 1;// C
        return n * factorial(n - 1);// C+T(n-1)
  }
```
Lo que busca este algoritmo es calcular la factorial de n, lo que se hace es descomponer a la factorial de n como n multiplicado de la factorial de n-1 y análogamente el de n-1 como n-1 por le factorial de n-2, así hasta llegar a la factorial de n-n ósea 0 lo que matemáticamente es 1, se resolverían las cadenas de multiplicaciones hasta que se llega al resultado factorial de n.

2.1.2
```
public boolean array220(int[] nums, int index) {
        if (index == nums.length - 1 || nums.length == 0)
              return false; // C1
        return nums[index] * 10 == nums[index + 1]
               || array220(nums, index + 1); // T(n)=C2+T(n-1)
  }
```
Este algoritmo retorna si en una matriz existen dos números consecutivos tal que el segundo sea 10 veces el primero esto se hace tomando en orden dos números de la matriz y comparándolos, este proceso se hará con cada pareja consecutiva existente hasta que se llegue al final de la matriz y si no se encuentra resultado se devuelve False.

2.1.3
```
public int count8(int n) {
        if (n == 0)// C1
              return 0;
        int val = n % 10 == 8 ? 1 : 0;// C2
        if (n % 100 == 88)// C3
              val = 2;// T(n)=C3+C4
        return val + count8(n / 10);// T(n)=C5+T(n-1)
```

PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

Este algoritmo dice cuantos 8 hay en una entero, lo hace mirando cifra de a dos cifras esto con el objetivo de que si se encuentran dos 8 juntos se dará un punto extra, ejemplo: 88 tendría 3 puntos, 2 puntos de los dos 8 y uno extra por estar juntos así el 808 tendrá menos puntaje que el 880. Así buscaríamos de a uno en uno, pero siempre observando el numero al lado para poder aumentar puntos extra.

2.1.4
```
public String stringClean(String str) {
        if (str.length() == 0 || str.length() == 1)
                return str;

        return str.charAt(0) == str.charAt(1) ? stringClean(str.substring(1, str.length()))
                        : str.substring(0, 1) + stringClean(str.substring(1, str.length()));
}
```

Este algoritmo quita los caracteres repetidos adyacentes ósea en la cadena “aab” retornaría “ab”, esto de la siguiente manera: si el String es mayor de longitud 1, si no es así, simplemente se retornaría el mismo String, ya que no existiría caracteres repetidos, luego en el caso recursivo, si el primer carácter es igual al segundo entonces se retornaría de nuevo el mismo método, pero sin contar con el primer carácter, en caso contrario, se retornaría el primer carácter más el método con el String empezando desde el segundo carácter de esta forma hasta que no se quiten todos los caracteres adyacentes repetidos no se avanzara en la cadena.

2.1.5
```
public static String changeXY(String str) {
        if (str.indexOf("x") == -1)
                return str;// C1

        return str.substring(0, str.indexOf("x") + 1).replace("x", "y")
                        + changeXY(str.substring(str.indexOf("x") + 1, str.length()));// T(n)=C2+T(n-1)
}
```

ChangeXY cambia las “x” en una cadena de caracteres por “y”, primero se buscan las “x” en la cadena para asegurarse de que hayan ya que si no lo hay se retornara la cadena, esto también sirve de caso base ya que en le caso recursivo lo que se hace es cortar la cadena en dos, la primera con la “x” encontrada la cual usaremos el método String.replace(OldChar char,NewChar char) para cambiarla por una “y”, con la segunda cadena se llamara recursivamente a Change ya así este proceso hasta que no queden “x”, al final de concatenan las cadenas y queda una con “y” en vez de “x” .

2.2 Recursividad 2

2.2.1
```
public static boolean groupSum5(int start, int[] nums, int target) {

        if (target == 0 && start == nums.length)
                return true;
        if (start == nums.length)
                return false;
        if (nums[start] % 5 == 0) {
                if (start + 1 != nums.length && nums[start + 1] == 1)
                        nums[start + 1] = 0;
                return groupSum5(start + 1, nums, target - nums[start]);
        }
        return groupSum5(start + 1, nums, target) || groupSum5(start + 1, nums, target - nums[start]);
}
```

groupSum5 lo que hace es responder a la pregunta si existe alguna suma de elementos en un arreglo dado que, de un objetivo determinado, pero esta es particular ya que debe cumplir ciertas especificaciones; todos los múltiplos de 5 deben ser parte de la suma y si hay un 1 inmediatamente después de un múltiplo de 5 no se contara. Hay dos casos de parada el primero es que se haya llegado al objetivo, pero con la condición de que se haya revisado todo el arreglo esto con la intención de no dejar pasar ningún múltiplo de 5 y el segundo caso de parada es que simplemente se hayan acabado los elementos de arreglo, se retornara True y False respectivamente. El proceso se simple, se revisa posición por posición creando un universo donde se añade a la suma y otro donde no, excepto a los múltiplos

PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

de 5 estos siempre se suman además cada vez que haya un múltiplo de 5 se revisa la posición siguiente y si es un 1 se remplaza este por un 0, y así se cumplen todas las condiciones y se crean todos los universos posibles para las sumas.

2.2.2
```
public boolean groupSumClump(int start, int[] nums, int target) {
        if (target == 0)
                return true;
        if (start == nums.length)
                return false;
        int cont = nums[start];
        while (start + 1 != nums.length && nums[start] == nums[start + 1]) {
                cont = nums[start] + cont;
                start++;
        }
```
Parecido al algoritmo anterior este también verifica si hay alguna suma que, del objetivo, pero esta también tiene una excepción, esta trata de que si hay numero consecutivos estos se deben estar juntos o están todos o no están todos, por lo que complica a la hora de la suma. Resolvimos estos impedimentos usando un ciclo que suma los valores consecutivos y al mismo tiempo avanza el contador de posición, como en el anterior nos movemos valor por valor, pero al usar los ciclos es como si nos desplazaramos un bloque entero. Igual que el anterior paramos cuando lleguemos a nuestro objetivo o nos quedemos sin valores.

2.2.3
```
public boolean groupNoAdj(int start, int[] nums, int target) {
        if (target == 0)
                return true;
        if (start>= nums.length)
                return false;
        return groupNoAdj(start + 2, nums, target - nums[start])
                        || groupNoAdj(start + 1, nums, target);
}
```
Análogamente a los anteriores es este algoritmo al determinar la existencia de alguna suma de elementos de un arreglo, pero este tiene la condición que si se suma un elemento no se podrá sumar el elemento inmediatamente después, así que simplemente se da la condición de que si el elemento se fuera a sumar no se movería un espacio sino 2 de esta manera el inmediatamente después se ignora. Las condiciones de parada son iguales que las 2 anteriores.

2.2.4
```
public boolean splitArray(int[] nums) {

        return auxiliar(nums, 0, 0);
}

public boolean auxiliar(int[] nums, int diferencia, int index) {
        if (index == nums.length && diferencia == 0)
                return true;
        if (index == nums.length)
                return false;
        return auxiliar(nums, diferencia - nums[index], index + 1)
                        || auxiliar(nums, diferencia + nums[index], index + 1);
}
```
Este algoritmo decir si existe una forma de dividir un arreglo de la forma que la suma de los elementos de cada grupo de igual, para hacerlo nos apoyamos en un método Auxiliar que se encarga de calcular los posibles grupos

PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

usando como referencia las posibles diferencias entre las sumas de cada grupo, de esta forma solo me tengo que fijar en la diferencia, de esta forma el método usando recurrencia creo los dos universos en una sumaría a la diferencia (Grupo 1) y en el otro restaría (Grupo 2), si cuando se acaben los elementos la diferencia es igual a 0 existirá un grupo igual, por otro lado si no da cero los grupos no serán iguales, se retornara True y False respectivamente.

2.2.5
```
public boolean splitOdd10(int[] nums) {
        return auxiliar(nums, 0, 0, 0);
    }

    public boolean auxiliar(int[] nums, int g1, int g2, int index) {
        if (index == nums.length && g1 % 10 == 0 && g2 % 2 == 1)
            return true;
        if (index == nums.length)
            return false;
        return auxiliar(nums, g1, g2 + nums[index], index + 1)
            || auxiliar(nums, g1 + nums[index], g2, index + 1);
    }
```
Este algoritmo decir si existe una forma de dividir un arreglo de la forma que la suma de los elementos de cada grupo tengan ciertas características, en grupo1debe ser múltiplo de 10 y el segundo debe ser impar, para hacerlo nos apoyamos en un método Auxiliar que se encarga de calcular los posibles grupos usando como referencia las posibles sumas de cada grupo, de esta forma solo me tengo que fijar las sumas, así el método usando recurrencia creando universos donde se suma cada elemento a un grupo o al otro, si cuando se acaben los elementos el grupo1 es múltiplo de 10 y el grupo2 sea impar retornara True, por otro lado retornara False.

3) Simulacro de preguntas de sustentación de Proyectos

3.1
```
public static int subCadena2(String str1, String str2, long len1, long len2){
    if(len1 == 0 || len2 == 0)
       return 0;
    if(str1.charAt((int)len1-1) == str2.charAt((int)len2-1))
       return 1 + subCadena2(str1, str2, len1-1, len2-1);
    else{
       int a = subCadena2(str1, str2, len1-1, len2);
       int b = subCadena2(str1, str2, len1, len2-1);
       return Math.max(a,b);
    }
  } Después de pasar a Wólfram Alpha:
```

$T(k) = C_3 (2^k - 1) + C_1 2^{k-1}$ (Donde c_1 es un parámetro arbitrario)
$T(k) = C_3 (2^k - 1) + C_1 2^{k-1}$
$T(k) = C_3 2^k - C_3 + C_1 2^{k-1}$
$T(k) = 2^k (C_3 + C/2) - C_3$
Al expresarlo con O:

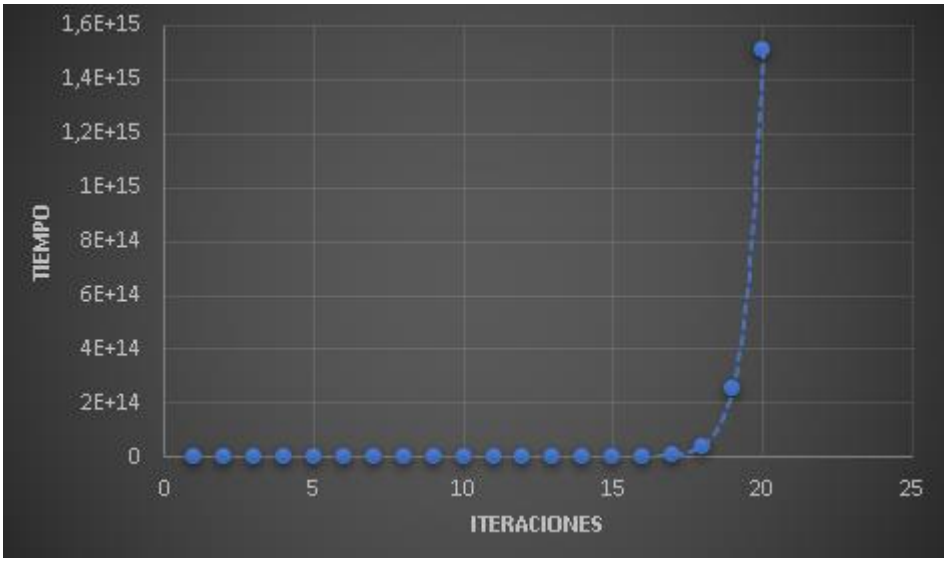
PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

$O(2^k (C_3 + C/2) - C_3) = O(2^k (C_3 + C/2))$ // Regla de la suma
$O(2^k)$ // Regla del producto
$O(2^k)$ (Con k = len1+len2, es decir las longitudes de ambas cadenas de caracteres)

3.2

Grafica de los datos obtenidos:



Ecuación de la línea de dispersión: $y = 0{,}6509e^{1{,}769x}$

3.3

Usando la ecuación anterior, se puede saber con completa certeza que este algoritmo duraria una cantidad muy grande de tiempo. Por lo que podemos notar que dicho algoritmo no sería muy eficaz a la hora de calcular la subsecuencia común más larga entre ADNs mitocondriales, puesto que toma una cantidad excesiva de tiempo para realizar dicho procedimiento, además de memoria.

3.4 groupSum5 lo que hace es responder a la pregunta si existe alguna suma de elementos en un arreglo dado que, de un objetivo determinado, pero esta es particular ya que debe cumplir ciertas especificaciones; todos los múltiplos de 5 deben ser parte de la suma y si hay un 1 inmediatamente después de un múltiplo de 5 no se contara.

3.5

3.5.1.1
```
public int factorial(int n) {
```

$$\textbf{\textit{Modelo}}: T(n) = \begin{cases} C_1, n = 1 \\ T(n) = C + T(n-1), n > 1 \end{cases}$$

$$\textbf{\textit{Ecucacion de Recurrencia}}: T(n) = C_2(n) + C_1$$

Calculo de Complejidad:

$O(C_2(n) + C_1)$ ---Por regla de Suma
$O(C_2(n))$ ----- Regla de Multiplicacion
$O(n)$

PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

3.5.1.2 `public boolean array220(int[] nums, int index) {`

$$Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + T(n-1), n > 0 \end{cases}$$
$$Ecucacion\ de\ Recurrencia: T(n) = C_2(n) + C_1$$
$$Calculo\ de\ Complejidad:$$
$$O(C_2(n) + C_1) \quad ----\text{Por regla de Suma}$$
$$O(C_2(n)) \quad -----\text{Regla de Multiplicacion}$$
$$O(n)$$

3.5.1.3 `public int count8(int n) {`

$$Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + T(n-1), n > 0 \end{cases}$$
$$Ecucacion\ de\ Recurrencia: T(n) = C_2(n) + C_1$$
$$Calculo\ de\ Complejidad:$$
$$O(C_2(n) + C_1) \quad ----\text{Por regla de Suma}$$
$$O(C_2(n)) \quad -----\text{Regla de Multiplicacion}$$
$$O(n)$$

3.5.1.4
```
public String stringClean(String str) {
}
```

$$Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + T(n-1), n > 0 \end{cases}$$
$$Ecucacion\ de\ Recurrencia: T(n) = C_2(n) + C_1$$
$$Calculo\ de\ Complejidad:$$
$$O(C_2(n) + C_1) \quad ----\text{Por regla de Suma}$$
$$O(C_2(n)) \quad -----\text{Regla de Multiplicacion}$$
$$O(n)$$

3.5.1.5 `public static String changeXY(String str) {`

$$Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + T(n-1), n > 0 \end{cases}$$
$$Ecucacion\ de\ Recurrencia: T(n) = C_2(n) + C_1$$
$$Calculo\ de\ Complejidad:$$
$$O(C_2(n) + C_1) \quad ----\text{Por regla de Suma}$$
$$O(C_2(n)) \quad -----\text{Regla de Multiplicacion}$$
$$O(n)$$

PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

3.5.2.1 `public boolean groupSum5(int start, int[] nums, int target) {`

$$\boldsymbol{Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + 2T(n-1), n > 0 \end{cases}}$$

$$\boldsymbol{Ecucacion\ de\ Recurrencia: T(n) = C_2(2^n - 1) + C_1 2^{n-1}}$$

Calculo de Complejidad:

$\boldsymbol{O(C_2(3^n - 1) + C_1 3^{n-1})}$ – – – –Por regla de Suma x2

$\boldsymbol{O(\frac{1}{2} C_2(3^n) + C_1 3^n)}$ – – – – – Regla de Multiplicacion x3

$\boldsymbol{O((3^n) + 3^n)}$

$\boldsymbol{O(2 \times 3^n)}$ – – – – – Regla de Multiplicacion

$\boldsymbol{O(3^n)}$ – – – – – Regla de Multiplicacion

3.5.2.2 `public boolean groupSumClump(int start, int[] nums, int target) {`

$$\boldsymbol{Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + 2T(n-1), n > 0 \end{cases}}$$

$$\boldsymbol{Ecucacion\ de\ Recurrencia: T(n) = C_2(2^n - 1) + C_1 2^{n-1}}$$

Calculo de Complejidad:

$\boldsymbol{O(C_2(2^n - 1) + C_1 2^{n-1})}$ – – – –Por regla de Suma

$\boldsymbol{O(C_2(2^n) + C_1 2^n)}$ – – – – – Regla de Multiplicacion

$\boldsymbol{O((2^n) + 2^n)}$

$\boldsymbol{O(2 \times 2^n)}$ – – – – – Regla de Multiplicacion

$\boldsymbol{O(2^n)}$

3.5.2.3 `public boolean groupNoAdj(int start, int[] nums, int target) {`

}

$$\boldsymbol{Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + T(n-1) + T(n-2), n > 0 \end{cases}}$$

$$\boldsymbol{Ecucacion\ de\ Recurrencia: T(n) = -C + C_1(F_n) + C_2 L_n}$$

Calculo de Complejidad:

$\boldsymbol{O(-C + C_1(F_n) + C_2 L_n)}$ – – – –Por regla de Suma

$\boldsymbol{O(C_1(F_n) + C_2 L_n)}$ – – – – – Regla de Multiplicacion x2

$\boldsymbol{O(F_n + L_n)}$

F_n is the n^{th} Fibonacci number

L_n is the n^{th} Lucas number

PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

3.5.2.4 `public boolean splitArray(int[] nums) {`

$$Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + 2T(n-1), n > 0 \end{cases}$$

$$Ecucacion\ de\ Recurrencia: T(n) = C_2(2^n - 1) + C_1 2^{n-1}$$

Calculo de Complejidad:

$O(C_2(2^n - 1) + C_1 2^{n-1})$ ----Por regla de Suma

$O(C_2(2^n) + C_1 2^n)$ ------ Regla de Multiplicacion

$O((2^n) + 2^n)$

$O(2 \times 2^n)$ ------ Regla de Multiplicacion

$O(2^n)$

3.5.2.4 `public boolean splitOdd10(int[] nums) {`

$$Modelo: T(n) = \begin{cases} C_1, n = 0 \\ T(n) = C + 2T(n-1), n > 0 \end{cases}$$

$$Ecucacion\ de\ Recurrencia: T(n) = C_2(2^n - 1) + C_1 2^{n-1}$$

Calculo de Complejidad:

$O(C_2(2^n - 1) + C_1 2^{n-1})$ ----Por regla de Suma

$O(C_2(2^n) + C_1 2^n)$ ------ Regla de Multiplicacion

$O((2^n) + 2^n)$

$O(2 \times 2^n)$ ------ Regla de Multiplicacion

$O(2^n)$

3. 6

En todos los casos se utiliza *n* como como variable de la complejidad, y en todos los casos representa en número de entradas respectivamente; en los algoritmos con string es el numero de caracteres de la cadena, en los de arreglos era el número de elementos del arreglo y siempre esta n se toma para el peor de los casos; ósea si buscáramos en una cadena con n caracteres carácter por carácter en el peor de los casos buscaríamos n veces, de eso trata n, representa el número de entradas para el peor de los casos (el total de entradas).

4) Simulacro de Parcial

4.1
1.A
2.C
3.C

4.2
1.B

4.3
1.B

4.4 *1.C*

4.5
1.A
2.B

4.5.1
1.C

4.6
1.0

PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

2.SumaAux (n, i+1)

5) Lectura recomendada (opcional)

Mapa conceptual

6) Trabajo en Equipo y Progreso Gradual (Opcional)

6.1 *Actas de reunión*
6.2 *El reporte de cambios en el código*
6.3 *El reporte de cambios del informe de laboratorio*

PhD. Mauricio Toro Bermúdez

Correo: | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473