

Primera Práctica

Individual de ADDA

Memoria técnica del proyecto

Roberto Camino Bueno, grupo asignado número 3

Índice:

- 1. Código con la soluciones de los problemas.**
- 2. Definir los tamaños y calcular los $T(n)$ para las distintas funciones implementadas considerando los casos mejor y peor.**
- 3. Volcado de pantalla con los resultados obtenidos en las pruebas realizadas.**

1. Código con la soluciones de los problemas.

P.I. 1. Ejercicio 57 en JAVA

// SOLUCIÓN FUNCIONAL EN JAVA 10:

```
public static String BuscaMayorCadenaMinusculaJava10(List<String> lista) {  
    return lista.stream()  
        .max(Comparator.comparing(x -> x.chars()  
            .filter(Character::isLowerCase).count()))  
        .get();  
}
```

El método obtiene el String con mayor número de caracteres tomando, de manera que toma cada carácter filtrando cada uno por los que sean minúsculas para contar cada minúscula.

// SOLUCIÓN ITERATIVA:

```
public static String BuscaMayorCadenaWhile(List<String> lista) {  
  
    String res = null;  
    Integer contRes = 0;  
    int j = 0;  
  
    while (j < lista.size()) {  
        if (cuentaMinusc(lista.get(j)).compareTo(contRes) > 0) {  
            res = lista.get(j);  
            contRes = cuentaMinusc(lista.get(j));  
        }  
        j++;  
    }  
    return res;  
}
```

Inicializamos el String resultado, el contador de minúsculas y un índice j que recorre cada String de la lista.

La condición de permanencia en el bucle permite recorrer la lista hasta el último String.

Dentro, tenemos una condición que llama al método auxiliar “cuentaMinusc” del elemento String por el que va el índice j y compara el contador devuelto con el contador del último String con mayor número de minúsculas.

En la primera iteración, contRes será 0 y entrará el primer String pero en los siguientes hasta el último se irá actualizando contRes así como el String resultado (res) .

Aumentamos el índice j para obtener los siguientes elementos de la lista.

Fuera del bucle ya devolvemos el último String actualizado.

//METODO SECUNDARIO ITERATIVO

```
private static Integer cuentaMinusc(String palabra) {  
  
    int i = 0;  
    Integer cont = 0;  
  
    while (i < palabra.length()) {  
        if (Character.isLowerCase(palabra.charAt(i))) {  
            cont++;  
        }  
        i++; // siguiente letra  
    }  
  
    return cont;  
}
```

Este método auxiliar devuelve el contador de minúsculas del String candidato.

// SOLUCIÓN RECURSIVA LINEAL:

```
public static String BuscaMayorCadenaRecursivo(List<String> lista) {
    return BuscaMayorCadenaRecursivoGeneral(0, 0, lista, null);
}

private static String BuscaMayorCadenaRecursivoGeneral(Integer j, Integer contRes,
List<String> lista, String res) {

    if (j >= lista.size()) {
        return res;

    } else {
        if (cuentaMinuscRecursivo(lista.get(j)).compareTo(contRes) > 0)
        {
            res = lista.get(j);
            contRes = cuentaMinuscRecursivo(lista.get(j));
        }
        j++;
        return BuscaMayorCadenaRecursivoGeneral(j, contRes, lista,
res);
    }

}
```

En el método público recibimos los valores e inicializamos las variables llamando al método privado.

El primer if asegura que si j-ésima llamada se pasa del tamaño, devuelve el String resultado.

En cambio, else, se encarga de calcular las llamadas recursivas usando el método auxiliar.

En cada nueva llamada recursiva hemos actualizado los valores de las variables.

//METODO SECUNDARIO O AUXILIAR RECURSIVO

```
public static Integer cuentaMinuscRecursivo(String palabra) {  
    return cuentaMinuscRecursivoGeneral(0, 0, palabra);  
}
```

```
private static Integer cuentaMinuscRecursivoGeneral(Integer i, Integer cont, String  
palabra) {
```

```
    if (i >= palabra.length()) { // si esta llamada se pasa del tamaño, devuelve  
res
```

```
        return cont;
```

```
    } else {
```

```
        if (Character.isLowerCase(palabra.charAt(i))) {  
            cont++;
```

```
        }
```

```
        i++; // siguiente letra
```

```
        return cuentaMinuscRecursivoGeneral(i, cont, palabra); //llamada  
recursiva
```

```
    }
```

```
}
```

Este método auxiliar funciona de manera recursiva similar al anterior y obtiene el contador de la palabra candidato.

// TEST

```
public static void main(String[] args) {
```

```
    List<String> lista1 = Arrays.asList("HolA", "12!.$%;.()", "&hoLá", "¡Resultado  
correcto!", "hOLA.", "holaa");
```

```
    List<String> lista2 = Arrays.asList("holaa", "HolA", "12!.$%;.()", "&hoLá",  
    "hOLA.", "¡Resultado correcto!");
```

```
    List<String> lista3 = Arrays.asList("¡Resultado correcto!", "HolA",  
    "12!.$%;.()", "holaa", "&hoLá", "hOLA.");
```

```
    System.out.println("Tomando un conjunto variado de listas como  
test."); System.out.println();
```

```
    System.out.println(lista1); System.out.println();
```

```
System.out.println(lista2);System.out.println();
System.out.println(lista3);
```

```
System.out.println("_____");
System.out.println();
```

```
String lista_res = BuscaMayorCadenaMinusculaJava10(lista1);
System.out.println("Resultado de la prueba en java 10: " + lista_res);
String lista_res2 = BuscaMayorCadenaWhile(lista1);
System.out.println("Resultado de la prueba con while: " + lista_res2);
String lista_res3 = BuscaMayorCadenaRecursivo(lista1);
System.out.println("Resultado de la prueba con recursivo: " + lista_res3);
```

```
System.out.println();
String lista_res4 = BuscaMayorCadenaMinusculaJava10(lista2);
System.out.println("Resultado de la prueba en java 10: " + lista_res4);
String lista_res5 = BuscaMayorCadenaWhile(lista2);
System.out.println("Resultado de la prueba con while: " + lista_res5);
String lista_res6 = BuscaMayorCadenaRecursivo(lista2);
System.out.println("Resultado de la prueba con recursivo: " + lista_res6);
```

```
System.out.println();
String lista_res7 = BuscaMayorCadenaMinusculaJava10(lista3);
System.out.println("Resultado de la prueba en java 10: " + lista_res7);
String lista_res8 = BuscaMayorCadenaWhile(lista3);
System.out.println("Resultado de la prueba con while: " + lista_res8);
String lista_res9 = BuscaMayorCadenaRecursivo(lista3);
System.out.println("Resultado de la prueba con recursivo: " + lista_res9);
```

```
}
```

Este método Test comprueba que mis anteriores algoritmos compruebas únicamente las letras que sean minúsculas y devuelve el String con mayor número de minúsculas sin importar la posición de la lista donde se encuentre.

P.I. 4. Ejercicio 64 en JAVA

// SOLUCIÓN ITERATIVA:

```
public static <T extends Comparable <? super T>> List<T>
fusionaListasWhile(List<T> l1, List<T> l2) {
    List<T> lfinal = new ArrayList<>();
    int a = 0; // índice l1
    int b = 0; // índice l2

    while (a < l1.size() || b < l2.size()) {

        if(a==l1.size()) {
            lfinal.add(l2.get(b));
            b++;
        }else if(b==l2.size()) {
            lfinal.add(l1.get(a));
            a++;
        }else if ((l1.get(a)).compareTo( l2.get(b)) < 0) { // e1 es
menor y meto su elemento
            lfinal.add(l1.get(a));
            a++;
        } else { // e2 es menor luego meto su elemento
            lfinal.add(l2.get(b));
            b++;
        }
    }
    return lfinal;
}
```

La utilidad de implementar `extends<T extends Comparable <? super T>>` es para que nuestro algoritmo resuelva comparaciones de tipo genérico heredando del tipo genérico T.

Inicializamos un lista genérica, y los índices de ambas listas.

La condición de permanencia en el bucle permite recorrer la lista hasta que ambos índices lleguen al último elemento de las listas.

Cuando un índice llega al final, se añaden todos los elementos de la otra lista sin distinción ya que están ordenadas por defecto. Mientras ambos índices no lleguen al

final de sus listas respectivas se realizará una comparación para meter el elemento menor y mantener un orden natural en la lista final.

// SOLUCIÓN RECURSIVA LINEAL:

```
public static <T extends Comparable <? super T>> List<T>
fusionaListasRecursivo(List<T> l1, List<T> l2) {
    return fusionaListasRecursivoGeneral(0, 0, l1, l2, new ArrayList<>());
}

private static <T extends Comparable <? super T>> List<T>
fusionaListasRecursivoGeneral(Integer a, Integer b, List<T> l1, List<T> l2, List<T>
lfinal) {

    if (a >= l1.size() && (b >= l2.size())) {
        return lfinal;
    } else {
        if(a==l1.size()) {
            lfinal.add(l2.get(b));
            b++;
        }else if(b==l2.size()) {
            lfinal.add(l1.get(a));
            a++;
        }else if ((l1.get(a)).compareTo( l2.get(b)) < 0) {
            lfinal.add(l1.get(a));
            a++;
        } else {
            lfinal.add(l2.get(b));
            b++;
        }
        return fusionaListasRecursivoGeneral(a, b, l1, l2, lfinal);
    }

}
```

En el método público recibimos los valores e inicializamos las variables llamando al método privado.

El primer if asegura que si los índices de las listas llegan al final y se pasan, devuelve la lista ordenada resultante.

En cambio, else, se encarga de calcular las llamadas recursivas con las mismas condiciones que tenía el while antes.
 En cada llamada recursiva irá actualizando cada índice hasta devolver por la rama del primer if la lista resultante.

// SOLUCIÓN FUNCIONAL EN JAVA 10:

```
public static <T extends Comparable <? super T>> List<Integer>
fusionaListasJava10(List<Integer> l1, List<Integer> l2) {
    Integer tam = l1.size()+l2.size();
    return IntStream
        .range(0,tam)
        //.filter(x-> x == l1.get(i) || x == l2.get(i) ) //este es el if
        //.filter(i-> l1.get(i)==l2.get(i))
        .boxed()
        collect(Collectors.toList());
}

/*
INTENTOS:

* ---- CONCAT Y SORTED, NO VÁLIDO POR COMPLEJIDAD AÑADIDA
return Stream.concat(l1.stream(), l2.stream())
    .sorted().collect(Collectors.toList());

* ---- USANDO TUPLAS
Stream<List<T>> stream1=Stream.of(l1);
Stream<List<T>> stream2=Stream.of(l2);
IntStream.range(0, l1.size()+l2.size());

    Pair.of(a, l1.get(a));
    Pair.of(b, l1.get(b));
    ...
    .boxed()
    .collect(Collectors.toList());

*---- DOS LISTAS A STREAM Y METER CADA ELEMENTO EN LIST
return Stream.of(l1,l2) //stream de listas
    .flatMap(x->x.stream())
    .filter( x->x
```

```

        .min(Comparator.naturalOrder()))
        .collect(Collectors.toList());
    */

// TEST

public static void main(String[] args) {

    List<Integer> l1 = Arrays.asList(1, 3, 4, 5);
    List<Integer> l2 = Arrays.asList(2, 3, 5, 6, 8, 10);

    List<String> l3 = Arrays.asList("AA", "AAA", "REPE", "ÚLTIMO");
    List<String> l4 = Arrays.asList("A", "REPE");

    List<Double> l5 = Arrays.asList(1.5, 3.5, 4.5, 5.0);
    List<Double> l6 = Arrays.asList(2.0, 3.5, 5.5, 6.2);

    System.out.println("Tomando un conjunto variado de listas como test:");
    System.out.println(l1);System.out.println(l2);System.out.println();
    System.out.println(l3);System.out.println(l4);System.out.println();
    System.out.println(l5);      System.out.println(l6);

    System.out.println("_____");System.out.println(
);

    System.out.println("Pruebas para listas tipo Integer:
");System.out.println();
    List<Integer> lista_res = fusionaListasJava10(l1, l2);
    System.out.println("Resultado de la prueba en java 10:    " + lista_res);
    List<Integer> lista_res2 = fusionaListasWhile(l1, l2);
    System.out.println("Resultado de la prueba while iterativa: " +
lista_res2);
    List<Integer> lista_res3 = fusionaListasRecurso(l1, l2);
    System.out.println("Resultado de la prueba recursiva:    " +
lista_res3);System.out.println();

    System.out.println("Pruebas para listas tipo String:
");System.out.println();
    List<String> lista_res5 = fusionaListasWhile(l3, l4);
    System.out.println("Resultado de la prueba while iterativa: " +
lista_res5);
    List<String> lista_res6 = fusionaListasRecurso(l3, l4);

```

```

        System.out.println("Resultado de la prueba recursiva: " +
lista_res6);System.out.println();

        System.out.println("Pruebas para listas tipo Double:
");System.out.println();
        List<Double> lista_res8 = fusionaListasWhile(l5, l6);
        System.out.println("Resultado de la prueba while iterativa: " +
lista_res8);
        List<Double> lista_res9 = fusionaListasRecursivo(l5, l6);
        System.out.println("Resultado de la prueba recursiva: " + lista_res9);
    }

}

```

P.I. 4. Ejercicio 57 en C

// SOLUCIÓN ITERATIVA:

```
string* BuscaMayorCadenaWhile(string_list lista) {  
  
    static string res; //cadena estática xq no cambiará muchas veces de valor y se  
    inicia a null  
    int contRes = 0; // lleva la cuenta de minusc del resultado  
    int j = 0;  
  
    while (j < lista.size ) {  
        if (cuentaMinusc(lista.data[j]) > contRes) {  
            strcpy(res, lista.data[j]); //copia en res este j-ésimo elemento  
            contRes = cuentaMinusc(lista.data[j]); // j-esimo contador  
        }  
  
        j++;    }  
    return &res; // devuelve dirección  
}
```

Este algoritmo iterativo funciona de manera similar al de java.

Inicializamos las variables igual pero res será un puntero que apunta a la dirección de memoria donde se encuentra el primer carácter de la palabra resultante con más caracteres en minúscula.

Para actualizar res usamos “strcpy()”.

Al final devolvemos la primera posición del carácter donde está la palabra con más caracteres en minúscula.

//METODO AUXILIAR ITERATIVO

```
int cuentaMinusc(string palabra) {  
  
    int i = 0;  
    int cont = 0;  
  
    while (i <= strlen(palabra)) {  
        char c = palabra[i];  
        if ('a' <= c && c <= 'z' ) {  
            cont++;  
        }  
        i++; // siguiente letra  
    }
```

```

    }

    return cont;
}

```

Este método auxiliar nos devuelve el contador de minúsculas de la palabra.

// SOLUCIÓN RECURSIVA LINEAL:

```

string* BuscaMayorCadenaRecursivo(string_list lista) {
    return BuscaMayorCadenaRecursivoGeneral(0, 0, lista);
}

string* BuscaMayorCadenaRecursivoGeneral(int j, int contRes, string_list lista) {
    static string res;
    if (j > lista.size) { // si esta llamada se pasa del tamaño, devuelve res
        return &res;
    } else {
        if (cuentaMinusc(lista.data[j]) > contRes) {
            strcpy(res, lista.data[j]);
            contRes = cuentaMinusc(lista.data[j]); // j-esimo contador
        }
        j++;
        return BuscaMayorCadenaRecursivoGeneral(j, contRes, lista);
    }
}
}

```

La versión recursiva del método es similar a java pero de nuevo devolviendo la dirección de memoria donde están los caracteres del string.

//METODO AUXILIAR RECURSIVO

```

int cuentaMinuscRecursivo(string palabra) {
    return cuentaMinuscRecursivoGeneral(0, 0, palabra);
}

int cuentaMinuscRecursivoGeneral(int i, int cont, string palabra) {

```

```

        if (i > strlen(palabra)) {
            return cont;

        } else {
            char c = palabra[i];
            if ('a' <= c && c <= 'z' ) {
                cont++;
            }
            i++;
            return cuentaMinuscRecursivoGeneral(i, cont, palabra);
        }

    }
}

```

Este método auxiliar nos devuelve el contador de manera recursiva.

//TEST

```

void pruebaIterativo1() {
    string l1[] = { "HolA", "12!!!!!!$%:.()", "&hoLç!", "¡Resultado correcto!",
    "hOLA.", "holaa" };
    string_list lprueba = create_string_list(l1, 6, 6); // (string * data, int tam, int size)
    printf("Resultado de la prueba en while es: %s \n",
        BuscaMayorCadenaWhile(lprueba));
    fflush(stdout);
}

```

```

void pruebaRecursivo1() {
    string l1[] = { "HolA", "12!!!!!!$%:.()", "&hoLç!", "¡Resultado correcto!",
    "hOLA.", "holaa" };
    string_list lprueba = create_string_list(l1, 6, 6); // (string * data, int tam, int size)
    printf("Resultado de la prueba recursiva es: %s \n",
        BuscaMayorCadenaRecursivo(lprueba));
    fflush(stdout);
}

```


P.I. 4. Ejercicio 64 en C

```
int_list fusionaListasWhile(int_list l1, int_list l2) {
    int_list lfinal = empty_int_list(l1.size + l2.size);
    int a = 0; // índice l1
    int b = 0; // índice l2

    while (a < l1.size || b < l2.size) {
        if(a==l1.size) {
            int w = l2.data[b];
            lfinal.data[lfinal.size] = w;
            lfinal.size++;
            b++;
        } else if(b==l2.size) {
            int x=l1.data[a];
            lfinal.data[lfinal.size] = x;
            lfinal.size++;
            a++;
        } else if (l1.data[a] < l2.data[b]) { // e1 es menor y meto
su elemento
            int y =l1.data[a];
            lfinal.data[lfinal.size] = y;
            lfinal.size++;
            a++;
        } else { // e2 es menor luego meto su elemento
            int z = l2.data[b];
            lfinal.data[lfinal.size] = z;
            lfinal.size++;
            b++;
        }
    }
    return lfinal;
}
```

El método iterativo tiene la misma estructura que en java variando en la sintaxis. Cuando comparamos los elementos de las listas lo asignamos a una variable local el menor y este va a parar a la lista final en su última posición que se va incrementando.

// SOLUCIÓN RECURSIVA LINEAL:

```
int_list fusionaListasRecursivo(int_list l1, int_list l2) {
    int_list lfinal = empty_int_list(l1.size + l2.size); //creo ya lfinal
    return fusionaListasRecursivoGeneral(0, 0, l1, l2, lfinal);
}

int_list fusionaListasRecursivoGeneral(int a, int b, int_list l1, int_list l2, int_list lfinal) {

    if (a >= l1.size && (b >= l2.size)) { // acaba y devuelve lo acumulado
        return lfinal;
    } else {
        if(a==l1.size) {
            int w = l2.data[b];
            lfinal.data[lfinal.size] = w;
            lfinal.size++;
            b++;
        } else if(b==l2.size) {
            int x=l1.data[a];
            lfinal.data[lfinal.size] = x;
            lfinal.size++;
            a++;
        } else if (l1.data[a] < l2.data[b]) { // e1 es menor y meto su
elemento
            int y =l1.data[a];
            lfinal.data[lfinal.size] = y;
            lfinal.size++;
            a++;
        } else { // e2 es menor luego meto su elemento
            int z = l2.data[b];
            lfinal.data[lfinal.size] = z;
            lfinal.size++;
            b++;
        }
        return fusionaListasRecursivoGeneral(a, b, l1, l2, lfinal);
    }
}

//llamar a la recursividad dentro del else
}
```

El método recursivo también tiene la misma estructura que en java y las condiciones funcionan igual que en el iterativo salvo que inicializo la lista final en el método recursivo público.

```
//TEST
```

```
void pruebaIterativo4() {
```

```
    int l1[] = { 1, 3, 5 };
```

```
    int l2[] = { 1, 2, 3, 4, 5 };
```

```
    int_list lprueba1 = create_int_list(l1, 3, 3); // (int * data, int tam, int size)
```

```
    int_list lprueba2 = create_int_list(l2, 5, 5); // (int * data, int tam, int size)
```

```
    int_list listaFusionada = fusionaListasWhile(lprueba1, lprueba2);
```

```
    printf("Resultado de la prueba en while es: \n");
```

```
    imprime_list_int(listaFusionada);
```

```
    fflush(stdout);
```

```
}
```

```
void pruebaRecursivo4() {
```

```
    int l1[] = { 1, 3, 5 };
```

```
    int l2[] = { 1, 2, 3, 4, 5 };
```

```
    int_list lprueba1 = create_int_list(l1, 3, 3); // (int * data, int tam, int size)
```

```
    int_list lprueba2 = create_int_list(l2, 5, 5); // (int * data, int tam, int size)
```

```
    int_list listaFusionada = fusionaListasRecursivo(lprueba1, lprueba2);
```

```
    printf("Resultado de la prueba recursiva es: \n");
```

```
    imprime_list_int(listaFusionada);
```

```
    fflush(stdout);
```

```
}
```

```
//MAIN
```

```
int main(void){

    printf("Para realizar test 1 o test 2 pulse ese nº y enter 2 veces \n\n");

    int numTest;
    scanf("%d",&numTest); //puntero apuntando al test que quiero, scanf para que lo
meta por teclado

    if(numTest == 1){
        printf("Test pi1: \n");
        pruebaIterativo1();
        pruebaRecursivo1();
    }

    if(numTest == 2){
        printf("Test pi4: \n");
        pruebaIterativo4();
        pruebaRecursivo4();
    }

    return 0;
}
```

Aquí elijo cual test probar introduciendo su número.

2. Definir los tamaños y calcular los $T(n)$ para las distintas funciones implementadas considerando los casos mejor y peor.

El tamaño del problema reside en el tamaño de la lista o listas que vamos a recorrer.

El orden de complejidad será la misma para cualquier lista ya que entrará en un proceso de comparación de elementos, entonces, no tendrá caso mejor y caso peor, siempre será de orden lineal.

En definitiva se trata de un problema con solución con complejidad de orden lineal.

P.I. 1. Ejercicio 57

• Iterativo:

- While: $n_{\text{while}} = \text{lista.size()} - j$. $j \equiv$ índice que recorre la lista

Se trata de una progresión aritmética ($n_j = \text{lista.size()} - j$, en cada iteración hay menos lista por recorrer y j es $j = j + 1$)

$T(n)_{\text{mejor}} = n + \phi$ y $T(n)_{\text{peor}} = n + K$ luego podemos concluir que

$T(n)_{\text{mejor}} = T(n)_{\text{peor}} = \Theta(n)$ ya que siempre tendrá una complejidad

de orden lineal $\Theta(n)$ dependiendo del tamaño de la lista:

$$T(n)_{\text{while}} = \sum_{f_i \in \text{pa}(\emptyset, 1)}^n 1 \cdot f_i^d \cdot \log^p f_i = \sum_{x \in \text{pa}(\emptyset, 1)}^n 1 \in \Theta(n)$$

- Método:

$n_{\text{BuscaMayorCadena}} = \text{lista.size()}$

$$T(n)_{\text{BuscaMayorCadena}} = n + 1 \Rightarrow \in \Theta(n)$$

La complejidad del método reside en el bucle, al tener que recorrer la lista entera será de orden $\Theta(n)$

- Recursivo: $n = \text{lista.size()} - j$; $\text{lista.size()} - (j+1) = k_0 - (j+1) = n-1$

Por tanto:

$$T(n)_{\text{BuscaMayorCadena}} = T(n-1) + 1 \in \underline{\Theta(n)}$$

El método en su versión recursiva tendrá la misma complejidad que en la iterativa; depende del tamaño de la lista: $\Theta(n)$ orden lineal.

P.I. 4. Ejercicio 64

- Iterativo:

- While: $n_{\text{while}} = l1.size() - a$ (tamaño). a = índice que recorre $l1$
Se trata de una progresión aritmética ($n_{a \in l1.size} = a++$)
Esto es similar para $l2.size() - b$ (tamaño para $l2$).

$$T(n)_{\text{mejor}} = n + \phi \quad \text{y} \quad T(n)_{\text{peor}} = n + k \quad \text{luego podemos concluir que}$$

$$\underline{T(n)_{\text{mejor}} = T(n)_{\text{peor}} = \Theta(n)}, \text{ siempre tiene una complejidad lineal que depende del tamaño de las listas.}$$

$$T(n)_{\text{while}} = \sum_{x \in \text{pa}(p, 1)}^n 1 \in \Theta(n)$$

- Método: $n_{\text{fusionalistas}} = l1.size() + l2.size()$

$$T(n)_{\text{fusionalistas}} = n + 1 = n \rightarrow \underline{\in \Theta(n)}.$$

El método tiene una complejidad lineal $\Theta(n)$ debido al bucle.

• Recursivo:

$$n = l1.size() - a ; n_1 = l1.size() - (a+1) = K_0 - (a+1) = n-1$$

Para $l2.size() - b$ es similar.

Por tanto:

$$T(n)_{\text{fusión de listas}} = T(n-1) + 1 \in \underline{\Theta(n)}$$

El método en su versión recursiva tendrá la misma complejidad lineal $\Theta(n)$ dependiendo del tamaño de las listas, igual que el iterativo.

3. Volcado de pantalla con los resultados obtenidos en las pruebas realizadas.

P.I. 1. Ejercicio 57 en JAVA

```
<terminated> ejercicio57 [Java Application] C:\Program Files (x86)\Java\jdk-11.0.1\bin\jav
Tomando un conjunto variado de listas como test:

[HoLa, 12!.$%;.()), &hoLá, ;Resultado correcto!, hOLA., holaa]

[holaa, HoLa, 12!.$%;.()), &hoLá, hOLA., ;Resultado correcto!]

[;Resultado correcto!, HoLa, 12!.$%;.()), holaa, &hoLá, hOLA.]

Resultado de la prueba en java 10: ;Resultado correcto!
Resultado de la prueba con while: ;Resultado correcto!
Resultado de la prueba con recursivo: ;Resultado correcto!

Resultado de la prueba en java 10: ;Resultado correcto!
Resultado de la prueba con while: ;Resultado correcto!
Resultado de la prueba con recursivo: ;Resultado correcto!

Resultado de la prueba en java 10: ;Resultado correcto!
Resultado de la prueba con while: ;Resultado correcto!
Resultado de la prueba con recursivo: ;Resultado correcto!
```

P.I. 4. Ejercicio 57 en JAVA

```
Problems @ Javadoc Declaration Console
<terminated> ejercicio64 [Java Application] C:\Program Files (x86)\Java\jdk-11.0.1\bin\javaw.exe (17 nov. 2018)
Tomando un conjunto variado de listas como test:
[1, 3, 4, 5]
[2, 3, 5, 6, 8, 10]

[AA, AAA, REPE, ÚLTIMO]
[A, REPE]

[1.5, 3.5, 4.5, 5.0]
[2.0, 3.5, 5.5, 6.2]

Pruebas para listas tipo Integer:

Resultado de la prueba en java 10: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Resultado de la prueba while iterativa: [1, 2, 3, 3, 4, 5, 5, 6, 8, 10]
Resultado de la prueba recursiva: [1, 2, 3, 3, 4, 5, 5, 6, 8, 10]

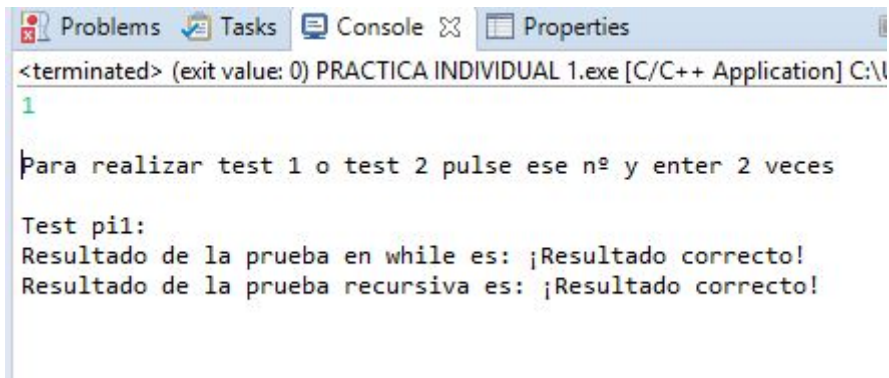
Pruebas para listas tipo String:

Resultado de la prueba while iterativa: [A, AA, AAA, REPE, REPE, ÚLTIMO]
Resultado de la prueba recursiva: [A, AA, AAA, REPE, REPE, ÚLTIMO]

Pruebas para listas tipo Double:

Resultado de la prueba while iterativa: [1.5, 2.0, 3.5, 3.5, 4.5, 5.0, 5.5, 6.2]
Resultado de la prueba recursiva: [1.5, 2.0, 3.5, 3.5, 4.5, 5.0, 5.5, 6.2]
```

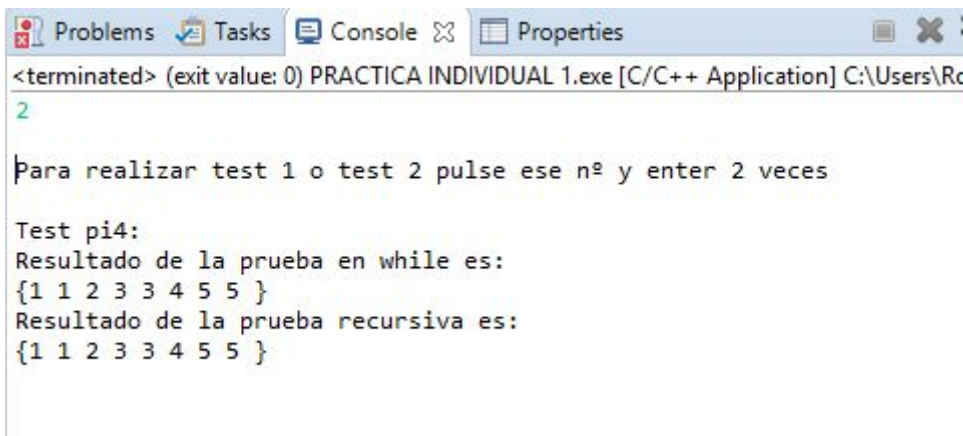

P.I. 1. Ejercicio 57 en C



```
Problems Tasks Console Properties
<terminated> (exit value: 0) PRACTICA INDIVIDUAL 1.exe [C/C++ Application] C:\I
1
Para realizar test 1 o test 2 pulse ese nº y enter 2 veces

Test pi1:
Resultado de la prueba en while es: ;Resultado correcto!
Resultado de la prueba recursiva es: ;Resultado correcto!
```

P.I. 4. Ejercicio 57 en C



```
Problems Tasks Console Properties
<terminated> (exit value: 0) PRACTICA INDIVIDUAL 1.exe [C/C++ Application] C:\Users\Rc
2
Para realizar test 1 o test 2 pulse ese nº y enter 2 veces

Test pi4:
Resultado de la prueba en while es:
{1 1 2 3 3 4 5 5 }
Resultado de la prueba recursiva es:
{1 1 2 3 3 4 5 5 }
```

El resultado en C será interactivo, es decir, cuando ejecutes el ‘main’ debes introducir qué problema quieres comprobar de manera que el 1 será el PI 2 y el 2 será el PI 2. Lo he implementado así porque el profesor quiere ambos problemas en el mismo proyecto, así no tengo dos ‘main’.