2-

La búsqueda binaria a dos niveles primero hace una búsqueda binaria en las columnas de la primera fila y después una búsqueda binaria en las filas de la columna seleccionada.

Búsqueda binaria en las columnas

En la primera parte, la función `busqueda_binaria_col` realiza una búsqueda binaria sobre las columnas, es decir la primera fila de la matriz. La búsqueda binaria tiene una complejidad logarítmica, por lo que para Nc columnas, el tiempo que toma es O(logNc).

La función de recurrencia para esta parte es:

$T_1(Nc)=T_1(Nc/2)+O(1)$

Resolviendo la recurrencia, nos queda:

$T_1(Nc)=O(\log Nc)$

Búsqueda binaria en las filas

Después de haber encontrado la columna, se hace la búsqueda binaria en las filas de esa columna, utilizando la función `busqueda_binaria_fil`. Dado que hay Nr filas en la matriz excluyendo la primera fila que contiene los identificadores de columnas, la búsqueda binaria en las filas tiene una complejidad O(logNr).

La función de recurrencia para la búsqueda en las filas es:

$T_2(Nr)=T_2(Nr/2)+O(1)$

Resolviendo esta recurrencia, obtenemos:

$T_2(Nr)=O(\log Nr)$

Complejidad total

Debido a que la búsqueda binaria en las filas se realiza después de la búsqueda binaria en las columnas, la complejidad total del algoritmo será la suma de ambas búsquedas. Por lo tanto, la complejidad total es:

$T(Nc,Nr)=O(\log Nc)+O(\log Nr)$

Finalmente, la complejidad del algoritmo de búsqueda binaria a dos niveles es:

$T(Nc,Nr)=O(\log Nc+\log Nr)$

Ejercicio 2: Obteniendo la formula de recurrencia para busqueda binaria de 2 niveles

Primero

```
public static int busqueda_binaria_fil(int A[][], int llave2, int linf,     ← T2
 int lsup, int col){
   if( linf == lsup ){
      if( A[linf][col] == llave2 ){
         System.out.println("Nivel 2: Llave "+llave2+" en posicion ("+
```



```
                           String.valueOf(linf)+","+String.valueOf(col)+")"
);
         return 1;
  }else{
         System.out.println("Nivel 2: Llave "+llave2+" no encontrada");
         return -1;
  }
   }else{
      int mid = linf + (lsup-linf)/2;
      if( llave2 <= A[ mid ][ col ] ){
         return busqueda_binaria_fil( A, llave2, linf, mid, col );
      }else{
         return busqueda_binaria_fil( A, llave2, mid+1, lsup, col );
      }
   }
}
public static int busqueda_binaria_col(int A[][], int llave1, int     ← T1
llave2, int linf, int lsup){
   if( linf == lsup ){
  if( A[0][linf] == llave1 ){
         System.out.println("Nivel 1: Llave "+llave1+" en columna "+
                                     String.valueOf(linf));
     return busqueda_binaria_fil( A, llave2, 1, A.length-1, linf );
  }else{
         System.out.println("Nivel 1: Llave "+llave1+" no encontrada");
         return -1;
  }
   }else{
      int mid = linf + (lsup-linf)/2;
      if( llave1 <= A[0][ mid ] ){
         return busqueda_binaria_col( A, llave1, llave2, linf, mid );
      }else{
         return busqueda_binaria_col( A, llave1, llave2, mid+1, lsup);
      }
   }
}
```

Para la busqueda binaria la complejidad $T_1(N_c)$ como el problema se divide a la mitad cada vez y llama a T_2

$\Rightarrow T_1(N_c) = T_1(\frac{N_c}{2}) + T_2(N_r) + O(1)$

En cuanto a la complejidad $T_2(N_r)$

$\Rightarrow T_2(N_r) = T_2(N_r/2) + O(1)$

Veamos que $T_2(N_r) = T_2\left(\frac{N_r}{2}\right) + d$

$\Rightarrow T_2(N_r) = T_2\left(\frac{N_r}{2}\right) + d = T_2\left(\frac{N_r}{4}\right) + 2d = \dots = T_2\left(\frac{N_r}{2^k}\right) + kd$

se detiene en $\frac{N}{2^k} = 1 \Rightarrow k = \log_2 N_r$

$\Rightarrow T_2(N_r) = T_2(1) + d\log_2(N_r)$ y como $T_2(1) \in O(1)$

$\Rightarrow$ para $c \geq d$ $T_2(N_r) \leq c\log_2(N_r) \Rightarrow T_2(N_r) \in O(\log N_r)$

para $T_1(N_c) = T_1\left(\frac{N_c}{2}\right) + T_1(N_r) + O(1)$

Como $T_2(N_r) \in O(\log N_r)$ entonces $T_1(N_c) = T_1\left(\frac{N_c}{2}\right) + O(\log N_r) + O(1)$

Sustituyendo $T_1(N_c) \leq T_1\left(\frac{N_c}{2}\right) + c_1 \log N_r + c_2$

$\Rightarrow T_1(N_c) \leq T_1\left(\frac{N_c}{2}\right) + c_1 \log N_r + c_2 \leq T_1\left(\frac{N_c}{4}\right) + 2c_1 \log N_r + 2c_2 \leq \dots \leq T_1\left(\frac{N_c}{2^k}\right) + kc_1 \log N_r + kc_2$

como $\frac{N_c}{2^k} = 1 \Rightarrow k = \log_2 N_c$

$\Rightarrow T_1(N_c) \leq T_1(1) + c_1 \log N_r \log_2 N_c + c_2 \log_2 N_c$

$T_1(1) \in O(1)$

$T_1(N_c) \leq O(1) + c_1 \log N_r \log_2 N_c + c_2 \log_2 N_c$

Sea $c \geq c_1$ y $c_3 \geq c_2$

$\Rightarrow T_1(N_c) \leq c\log_2 N_c \log N_r + c_3 \log_2 N_c$

$\Rightarrow T_1(N_c) \in O(\log N_c + \log N_r)$

3-

Condición base: Cuando la suma $k_1 + k_2$ sea igual a n, se tiene que imprimir la combinación y terminamos esa rama de la recursión.

Recursión: Todo valor de k_1 (De o a n), k_2 se va a calcular como $k_2 = n - k_1$, de esta forma $k_1 + k_2 = n$.

```java
public class CombinacionesSuma {

    // Función recursiva para imprimir todas las combinaciones de k1 y k2 que suman n
    public static void combinaciones_suma(int k1, int n) {
        // La variable k2 es calculada como n - k1
        int k2 = n - k1;

        // Imprimimos la combinación cuando k1 + k2 == n
        System.out.println("(" + k1 + ", " + k2 + ")");

        // Condición de parada: si k1 es menor que n, hacemos la llamada recursiva
        if (k1 < n) {
            combinaciones_suma(k1 + 1, n); // Incrementamos k1 y seguimos recursivamente
        }
    }

    public static void main(String[] args) {
        int n = 5; // Ejemplo: número que queremos descomponer
        combinaciones_suma(0, n); // Llamada inicial con k1 = 0
    }
}
```

Código:

La función combinaciones_suma(int k1, int n):

- Recibe dos parámetros: k1 es el valor inicial que se va incrementando en cada llamada recursiva, y n es el valor total que queremos descomponer.
- Va a calcular el valor de k_2 como $k_2=n-k_1$.
- Imprime la combinación (k_1,k_2).
- Si k_1 aún no alcanza n, llama a sí misma con $k_1 + 1$, esto va a generar la siguiente combinación.
- Llamada inicial : Desde main, llamamos a la función con $k_1=0$ y el valor de n.

Ejemplo de salida:

Para n=5, la salida sería:

(0, 5)

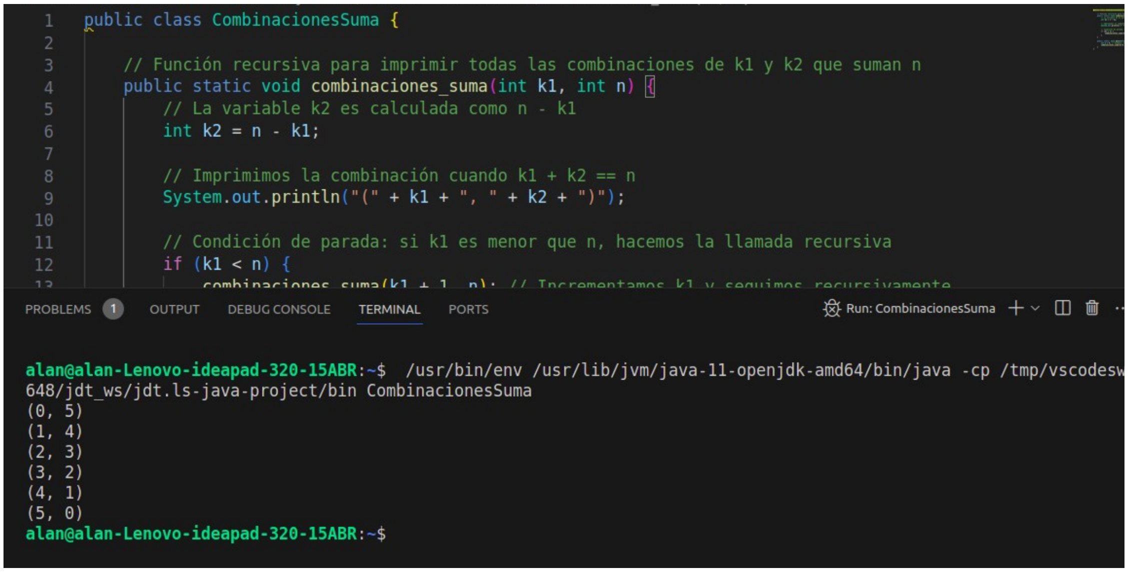
(1, 4)

(2, 3)

(3, 2)

(4, 1)

(5, 0)


public class CombinacionesSuma {
// Función recursiva para imprimir todas las combinaciones de k1 y k2 que suman n
public static void combinaciones_suma(int k1, int n) {
// La variable k2 es calculada como n - k1
int k2 = n - k1;
// Imprimimos la combinación cuando k1 + k2 == n
System.out.println("(" + k1 + ", " + k2 + ")");
// Condición de parada: si k1 es menor que n, hacemos la llamada recursiva
if (k1 < n) {
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
Run: CombinacionesSuma
alan@alan-Lenovo-ideapad-320-15ABR:~$ /usr/bin/env /usr/lib/jvm/java-11-openjdk-amd64/bin/java -cp /tmp/vscodesw
648/jdt_ws/jdt.ls-java-project/bin CombinacionesSuma
(0, 5)
(1, 4)
(2, 3)
(3, 2)
(4, 1)
(5, 0)
alan@alan-Lenovo-ideapad-320-15ABR:~$