Código completo

```c
typedef struct {
    int nota; // Armazena a nota com menor distância em relação à anterior
    int dist; // Armazena a distância entre as duas notas
} notaDistancia;

void calculaDist(notaDistancia *menor_nota, int* lst, int index, int size, bool last){
    int distancia;

    // Encontra a distância e insere na matriz
    if (!last) {
        distancia = lst[index] - lst[index - 1];
        if (distancia < menor_nota->dist) {
            menor_nota->dist = distancia;
            menor_nota->nota = lst[index];
        }
    } else {
        distancia = size - (lst[index] - lst[0]); // calcula a maior distância entre a primeira e últim
        if (distancia < menor_nota->dist) {
            menor_nota->dist = distancia;
            menor_nota->nota = lst[0];
        }
    }
}

int* recebeListaNotas(int tam_acorde, int tam_escala, notaDistancia *menor_nota){
    int *lst = (int*) malloc(tam_acorde * sizeof(int));
    for (int i = 0; i < tam_acorde; i++) {
        scanf("%d", &lst[i]);

        // Insere as distâncias entre cada nota na matriz de distâncias
        if (i) calculaDist(menor_nota, lst, i, tam_escala, false);
    }
    // Insere a distância entre a primeira e a última nota
    calculaDist(menor_nota, lst, tam_acorde - 1, tam_escala, true);

    return lst;
}

void freeM(int **m, int size){
    for (int i = 0; i < size; i++)
        free(m[i]);
    free(m);
}

void copiaLista(int *l1, int *l2, int size){
    for (int i = 0; i < size; i++) l1[i] = l2[i];
}

int binSearch(int *lst, int x, int start, int end){ // Supomos previamente que é garantido que o valor
    if (start <= end) {
        int idx = start + (end - start) / 2;
```

```c
        if (lst[idx] == x) return idx;
        if (lst[idx] > x) return binSearch(lst, x, start, idx - 1);
        else return binSearch(lst, x, idx + 1, end);
    }
}

int linSearch(int *lst, int x, int size){
    for (int i = 0; i < size; i++)
        if (lst[i] >= x) return i;
}

// Funções principais
int testaPossibilidades(int **blocos, int *lista_notas, int tam_escala, int tam_acorde, int tam_blocos,
    int it = tam_blocos,
        falhas = 0;

    // Repete um número de vezes igual ao total de possibilidades de divisão
    while (it--) {
        // printf("%d\n\n", it); // DEBUG
        int *t_lst = (int*) malloc(tam_acorde * sizeof(int)); // Criamos uma lista que vai ser alterada
        int k = 0, // Referencial da nota (geral) que vai ser inserida no bloco
            lst_index; // Referencial da nota que vai ser analisada na lista de notas do acorde

        copiaLista(t_lst, lista_notas, tam_acorde);
        // printf("\n\n-- %d --\n", nota_dist_min); // DEBUG

        if (falhas == 0) lst_index = binSearch(t_lst, nota_dist_min, 0, tam_acorde);
        else lst_index = linSearch(t_lst, nota_dist_min, tam_acorde);

        // For-loop principal
        for (int i = 0; i < tam_acorde; i++) { // Máximo de divisões que devem ser feitas
            int retiradas = 0;
            for (int j = 0; j < tam_blocos; j++) { // Quantidade de notas que cabem em uma divisão
                int nota;
                // Correção caso cheguemos ao fim da escala
                if (nota_dist_min + k >= tam_escala) nota = (nota_dist_min + k) - tam_escala;
                else nota = nota_dist_min + k;

                blocos[i][j] = nota;
                k++; // Vai para a próxima nota

                if (blocos[i][j] == t_lst[lst_index]) {
                    retiradas++;
                    if (retiradas > 1) break; // Sai do loop antes de atualizar o índice

                    lst_index++; // "Retiramos" a nota da lista de notas temporária e atualizamos o índ
                    if (lst_index == tam_acorde) lst_index = 0;
                }
            }

            // Ou duas notas estão no mesmo bloco, ou nenhuma nota foi inserida
            if (retiradas == 0 || retiradas > 1) {
                falhas++;
                break;
```

```c
            }
        }
        // Atualiza o valor inicial
        if (nota_dist_min + 1 >= tam_escala) nota_dist_min = 0;
        else nota_dist_min++;
        if (falhas == 0) break;

        free(t_lst);
    }
    return falhas == tam_blocos;
}

// Retorna se é possível fazer a divisão ou não
bool divide(int tam_escala, int tam_acorde, int *lista_notas, int nota_dist_min){
    // Possibilidade de divisão
    int tam_blocos = tam_escala / tam_acorde;
    int **divisoes = (int**) malloc(tam_acorde * sizeof(int*));
    for (int i = 0; i < tam_acorde; i++)
        divisoes[i] = (int*) malloc(tam_blocos * sizeof(int));

    int final = testaPossibilidades(divisoes, lista_notas, tam_escala, tam_acorde, tam_blocos, nota_dis
    freeM(divisoes, tam_acorde);
    return final == 1 ? false : true;
}

// Saída
void pExit(bool possivel){
    if (!possivel) printf("N\n");
    else printf("S\n");
}

int main(){
    int tam_escala, tam_acorde;
    scanf("%d %d", &tam_escala, &tam_acorde);

    notaDistancia menor_nota;
    menor_nota.dist = tam_escala;
    // Segunda linha de entrada (recebe a entrada e preenche a matriz de distâncias)
    int *lista_notas = recebeListaNotas(tam_acorde, tam_escala, &menor_nota);

    bool resultado = divide(tam_escala, tam_acorde, lista_notas, menor_nota.nota);
    pExit(resultado);

    free(lista_notas);

    return 0;
}
```

Análise das funções individuais

Tomando n = quantidade de notas do acorde,

calculaDist

```
int distancia;

if (!last) {                                  <- # = 1
    distancia=lst[index]-lst[index-1];        <- # = 1
    if (distancia < menor_nota->dist) {       <- # = 1
        menor_nota->dist=distancia;           <- # = 1
        menor_nota->nota=lst[index];          <- # = 1
    }
} else {                                      <- # = 1
    distancia=size-(lst[index]-lst[0]);       <- # = 1
    if (distancia < menor_nota->dist) {       <- # = 1
        menor_nota->dist=distancia;           <- # = 1
        menor_nota->nota=lst[0];              <- # = 1
    }
}
```

Custo máximo $= 7$ (if não é cumprido, logo entra no else) $\Longrightarrow C(n) = O(1)$

recebeListaNotas

```
int *lst=malloc(tam_acorde*sizeof(int));      <- # = 1
for (int i=0; i < tam_acorde; i++) {          <- # = i
    scanf("%d", &lst[i]);                    <- # = i * 1

    if (i) calculaDist(menor_nota,lst,i,tam_escala,false); <- # = i * Custo da função
}
calculaDist(menor_nota,lst,tam_acorde-1,tam_escala,true); <- # = 7

return lst;                                   <- # = 1
```

Encontrando o custo do laço, temos

$$C(n) = \sum_{i=0}^{n-1} 1 + 7 = 8(n-1) = 8n - 8.$$

Somando a quantidade de repetições das outras linhas do algoritmo,

$$C(n) = 1 + (8n - 1) + 7 + 1 = 8n + 1.$$

Como o resto da função tem custo constante, logo $C(n) = O(n)$.

freeM

```
for (int i=0; i < size; i++)                 <- # = i
    free(m[i]);                              <- # = i * 1
free(m);                                     <- # = 1
```

Tomamos $c \in \mathbb{N}^* |\ c =$ constante que, multiplicada por n, nos dá o tamanho da escala. Isso é possível pela própria restrição do exercício, que afirma que o tamanho da escala é um múltiplo do tamanho do acorde. De forma semelhante à função anterior, temos

$$C(n) = 1 + \sum_{i=0}^{c(n-1)} 1 = c(n-1) + 1 = cn - c + 1.$$

que nos dá o custo $C(n) = O(n)$.

copiaLista

```
for (int i=0; i < size; i++)                <- # = i
    l1[i] = l2[i];                          <- # = i * 1
```

Tomando c igual à função anterior, temos

$$C(n) = \sum_{i=0}^{c(n-1)} 1 = c(n-1) = cn - c.$$

Logo, $C(n) = O(n)$.

binSearch

```
if (start<=end) {
    int idx=start+(end-start)/2;

    if (lst[idx]==x) return idx;
    if (lst[idx] > x) return binSearch(lst, x, start, idx-1);
    else return binSearch(lst, x, idx+1, end);
}
```

Sabemos que o custo da busca binária é $O(log\ n)$ ### **linSearch**

```
for (int i=0; i < size; i++)
    if (lst[i]>=x) return i;
```

Sabemos que o custo da busca linear é $O(n)$

testaPossibilidades

Primeiramente, tomamos o tamanho da escala como $M \mid M \in \mathbb{N},\ M > 3$

```
int it=tam_blocos,                        <- # = 1
    falhas=0;                             <- # = 1

while (it--) {                            <- # = it
    int *t_lst=malloc(tam_acorde*sizeof(int)); <- # = 1
    int k=0,                              <- # = 1
        lst_index;                        <- # = 1

    copiaLista(t_lst,lista_notas,tam_acorde); <- # = laço * Custo da função

    if (falhas==0)                        <- # = 1
        lst_index=binSearch(t_lst, nota_dist_min, 0, tam_acorde); <- # = log n (esta linha só se repete
    else
        lst_index=linSearch(t_lst, nota_dist_min, tam_acorde); <- # = (it-1)*Custo da função

    for (int i=0; i < tam_acorde; i++) { <- # = i
        int retiradas=0;                  <- # = 1
        for (int j=0; j < tam_blocos; j++) { <- # = j
            int nota;                     <- # = 1 * j

            if (nota_dist_min+k>=tam_escala) <- # = 1 * j
```

```
                nota=(nota_dist_min+k)-tam_escala; <- # = 1
            else
                nota=nota_dist_min+k;        <- # = 1

            blocos[i][j]=nota;               <- # = 1 * j
            k++;                             <- # = 1 * j

            if (blocos[i][j]==t_lst[lst_index]) { <- # = 1 * j
                retiradas++;                 <- # = 1
                if (retiradas > 1)           <- # = 1
                    break;                   <- # = 1

                lst_index++;                 <- # = 1
                if (lst_index==tam_acorde)   <- # = 1
                    lst_index=0;             <- # = 1
            }
        }


        if (retiradas==0 || retiradas > 1) { <- # = 1 * i
            falhas++;                        <- # = 1
            break;                           <- # = 1
        }
    }
    if (nota_dist_min+1>=tam_escala)         <- # = 1 * it
        nota_dist_min=0;                     <- # = 1
    else
        nota_dist_min++;                     <- # = 1
    if (falhas==0)                           <- # = 1 * it
        break;                               <- # = 1

    free(t_lst);                             <- # = 1 * it
}
return falhas==tam_blocos;                   <- # = 1
```

No pior caso, a divisão ideal será encontrada a partir da segunda iteração, ou seja, o algoritmo não atinge o critério de parada definido em

```
if (falhas==0)
        break;
```

que encerra o laço while na primeira iteração.

A função de custo será dada por

$$C(n) = C(it) + 3.$$

Como as linhas internas dos laços têm custo constante, logo

$$\begin{aligned} C(j) &\approx \sum_{j=0}^{\frac{M}{n}-1} 5 = 5\left(\frac{M}{n} - 1\right) \\ &= \frac{5M}{n} - 5, \end{aligned}$$

e, assim,

$$
\begin{aligned}
C(i) \approx \sum_{i=0}^{n-1} 2 + C(j) \approx (n-1)\left(\frac{5M}{n} - 5\right) \\
= 5M - 5n - \frac{5M}{n} + 1.
\end{aligned}
$$

Logo,

$$
\begin{aligned}
C(it) &\approx O(log\ n) + \sum_{it=0}^{\frac{M}{n}-1} 7 + O(n) + O(n) + C(i) \\
&= M\left(\frac{5M}{n} - \frac{M}{n^2} + \frac{9}{n} - \frac{2 \cdot O(N)}{n}\right) - 2 \cdot O(n) + 5n.
\end{aligned}
$$

Pela definição do próprio exercício, podemos expressar da seguinte forma

$$M = cn,$$

como explicamos na análise da fução **freeM**. Substituindo na equação anterior, temos

$$
\begin{aligned}
C(it) &\approx cn\left(\frac{5cn}{n} - \frac{cn}{n^2} + \frac{9}{n} - \frac{2 \cdot O(N)}{n}\right) - 2 \cdot O(n) + 5n \\
&= 5c^2n - c^2 + 9c - 2 \cdot O(n) - 2 \cdot O(n) + 5n.
\end{aligned}
$$

Representando $O(n) = c_k n,\ c_k \in \mathbb{R}_+^*$, podemos afirmar

$$5c^2n - c^2 + 9c - 2 \cdot O(n) - 2 \cdot O(n) + 5n = 5c^2n - c^2 + 9c - 2c_0n - 2c_1n + 5n.$$

Como c, c_0 e c_1 são constantes, podemos reescrever da seguinte forma

$$5c^2n - c^2 + 9c - 2c_0n - 2c_1n + 5n = c_{k0}n + c_{k1} + c_{k2}n + c_{k3}n + c_{k4}n,$$

em que $c_{k0} = 5c^2,\ c_{k1} = 9c - c^2,\ c_{k2} = -2c_0,\ c_{k3} = -2c_1$ e $c_{k4} = 5$. Assim,

$$
\begin{aligned}
c_{k0}n + c_{k1} + c_{k2}n + c_{k3}n + c_{k4}n &= n(c_{k0} + c_{k2} + c_{k3} + c_{k4}) + c_{k1} \\
&= c_{k5}n + c_{k1},
\end{aligned}
$$

em que $c_{k5} = c_{k0} + c_{k2} + c_{k3} + c_{k4}$. Pela expressão, podemos intuitivamente afirmar que $C(it) = O(n)$, logo

$$
\begin{aligned}
C(n) &= O(n) + 3 \\
&= O(n).
\end{aligned}
$$

divide

```
int tam_blocos = tam_escala/tam_acorde;          <- # = 1
int **divisoes = malloc(tam_acorde*sizeof(int*)); <- # = 1
for (int i=0; i < tam_acorde; i++)               <- # = i
    divisoes[i] = malloc(tam_blocos*sizeof(int));

int final=testaPossibilidades(divisoes, lista_notas, tam_escala, tam_acorde, tam_blocos, nota_dist_min)
freeM(divisoes, tam_acorde);                     <- O(n)
return final==1 ? false : true;                  <- # = 1
```

O laço for terá custo O(n), uma vez que se repete n vezes apenas. Assim, como a função apenas possui constantes ou funções com ordem linear, podemos afirmar que $C(n) = O(n)$.

pExit

```
if (!possivel)
    printf("N\n");
else
    printf("S\n");
```

Claramente, temos uma função com custo $C(n) = O(1)$.

main

```
int tam_escala, tam_acorde;                      <- # = 1
scanf("%d %d", &tam_escala, &tam_acorde);        <- # = 1

notaDistancia menor_nota;                        <- # = 1
menor_nota.dist=tam_escala;                      <- # = 1

int *lista_notas=recebeListaNotas(tam_acorde,tam_escala,&menor_nota); <- O(n)

bool resultado=divide(tam_escala,tam_acorde,lista_notas,menor_nota.nota); <- O(n)
pExit(resultado);                                <- O(1)

free(lista_notas);                               <- # = 1

return 0;                                        <- # = 1
```

De forma semelhante à função **divide**, temos apenas constantes ou funções com custos lineares. Assim, o custo também será $C(n) = O(n)$.

Portanto, podemos afirmar que a **ordem de complexidade do algoritmo completo** é $O(n)$.