TPI2 ATR++

6 de noviembre de 2019

Castro Russo Matias Nahuel, 203/19

Yazlle Maximo, 310/19

Aclaración para todos los ejercicios:

En todos los casos, k es constante y el subíndice es para en principio indicar que provienen de distintas instrucciones y que no necesariamente son iguales, lo denotamos haciendo referencia a el çosto", que exactamente no lo conocemos, pero sabemos que es constante (en los casos que es k), el cual no esta en función de n. Finalmente metemos todas las k en la misma bolsa de constantes, ya que cualquier operación entre constantes es constante y las mismas no hacen a la complejidad.
En el caso de las instrucciones que hacen mas de una operaciones también constantes, las agrupamos en un solo k subíndice. Por ejemplo en los for, al decir tal constante nos estamos refiriendo a el "*costo*" de inicializar la variable iterativa, hacer la comparación de la condición para que entre o no al for y en cada iteración aplicarle la operación correspondiente. En el caso de las k propias del for, en los siguientes ejercicios es posible que no lo aclaremos explicitamente, sabemos que siempre esta ahí, aunque no haga a la complejidad del algoritmo :)
Por n siempre nos estamos refiriendo a la longitud del audio a.
Estas mismas aclaraciones corren para todos los calculos de este documento.
En todos los ejercicios las funciones auxiliares que reciben vectores de algún tipo como parametro, lo hacen por referencia
Redundante, pero en todos nos estamos refiriendo y calculando en base al peor caso.

Ejercicio 2.1

revertirAudio

```
1 audio revertirAudio(audio a, int canal, int profundidad) {
2     audio res = a;                                                    //k_{1} *|a|
3     for (int block = 0; block < a.size()/canal ; ++block) {          //sum y k_{2}
4         for (int j = 0; j < canal ; ++j) {                           //sum y k_{3}
5             res[canal*block + j] = a[a.size() - canal*(block+1) + j];   //k_{4}
6         }
7     }
8     return res;                                                       //k_{5}*|a|
9 }
```

$\#O_{revertirAudio} : k_1 * n + (\sum_{i=0}^{\frac{|a|}{canal}-1} k_2 + (\sum_{i=0}^{canal-1} k_3 + k_4)) + k_5 * |a|$

$\Rightarrow T_{revertirAudio}(n) = k_1 * n + (\sum_{i=0}^{\frac{n}{canal}-1} k_2 + (\sum_{i=0}^{canal-1} k_3 + k_4)) + k_5 * n$

$\Rightarrow T_{revertirAudio}(n) = k_1 * n + (\frac{n}{canal}) * k_2 * (canal) * (k_3 + k_4) + k_5 * n$

$\Rightarrow T_{revertirAudio}(n) = k_1 * n + n * k_2 * (k_3 + k_4) + k_5 * n$

$\Rightarrow T_{revertirAudio}(n) = n * (k_1 + k_2 + (k_3 * k_4) + k_5)$

$\Rightarrow T_{revertirAudio}(n) = n * (k) \in \mathcal{O}(n)$

limpiarAudio

```
1 void limpiarAudio(audio& a, int profundidad, vector<int>& outliers) {
2     audio a0Sort=a;                                  //k*n
3     selectionSort(a0Sort);                           //k*(n*n)
4     int percentil= getPercentil95(a0Sort);           //k
5     setOutliersPositions(a,outliers,percentil);      //k*n
6     modifyOnlyOutliers(a,outliers,percentil);        // k*(n*n)
7 }
```

$\#O_{limpiarAudio} : k * n + \#O_{selectionSort} + (k + \#O_{getPercentil95}) + \#O_{setOutliersPositions} + \#O_{ModifyOnlyOutliers}$

$\Rightarrow T_{limpiarAudio}(n) = k * n + k * n^2 + (k + k) + k * n + k * n^2$

$\Rightarrow T_{limpiarAudio}(n) = k * (n^2 + n) \in \mathcal{O}(n^2)$

Auxiliares de limpiarAudio

selectionSort

```
void selectionSort (audio &a){
    for (int j = 0; j < a.size() ; ++j) {
        int minimunPosition = getMinimumPositionFrom(a,j);     //k_{0}+k*n
        int swapAux = a[j];                                    //k_{1}
        a[j]=a[minimunPosition];                               //k_{2}
        a[minimunPosition]=swapAux;                            //k_{3}
    }
}
```

$\#O_{selectionSort} : \sum_{j=0}^{|a|-1}(k_{sumatoria} + (k_0 + \#O_{getMinimumPositionFrom}) + k_1 + k_2 + k_3)$

$\Rightarrow T_{selectionSort}(n) = n * (k_{sumatoria} + (k_0 + k * n) + k_1 + k_2 + k_3)$

$\Rightarrow T_{selectionSort}(n) = n * (k_{sumatoria} + (k * n) + k) = n * (k + (k * n))$

$\Rightarrow T_{selectionSort}(n) = n * (k * n) = k * n^2 \in \mathcal{O}(n^2)$

getMinimumPositionFrom

```
int getMinimumPositionFrom (audio &a,int &from){
    int minimumPositionCandidate=from;                 //k_{0}
    for (int i = from; i < a.size() ; ++i) {
        if(a[i]<a[minimumPositionCandidate]){          //k_{1} + k_{2} + k_{3}
            minimumPositionCandidate=i;                //k_{4}
        }
    }
    return minimumPositionCandidate;                   //k_{5}
}
```

$\#O_{selectionSort} : k_0 + (\sum_{i=from}^{|a|-1} k_{for} + k_1 + k_2 + k_3 + k_4) + k_5$

$\Rightarrow T_{selectionSort}(n) = k + (\sum_{i=from}^{n-1} k) + k$

$\Rightarrow T_{selectionSort}(n) = k + (k' * n) + k = k * n \in \mathcal{O}(n)$

En criollo, lo que hace la interación es iterar una subsecuencia desde la posición from, hasta n-1. Obviamente $0 \leq$ from $< n$, por lo tanto en el fondo from va dependiendo de n, y al ser una subsecuencia termino iterando k*n veces

getPercentil95

```
int getPercentil95 (audio &a){
    int percentil=0;                                    //k_{0}
    percentil= a[int(0.95*a.size())];                   //k_{1} + k_{2} + k_{3}
    return percentil;                                   //k_{4}
}
```

$\#O_{getPercentil95} : k_0 + k_1 + k_2 + k_3 + k_4$

$\Rightarrow T_{getPercentil95} = k \in \mathcal{O}(1)$

Usamos que la complejidad de v.size() para el tipo vector<algunTipo>es O(1), ya que por la estructura de dato con la cual esta armada el tipo vector, ademas de las unidades donde guarda los elementos del mismo, almacena una unidad de memoria en la cual guarda la longitud del mismo. Por lo tanto el costo de acceder a esa información es constante.

setOutliersPositions

```
void setOutliersPositions (audio &a,vector<int> &outliers, int percentil){
    outliers={};                                                    //k_{0}
    for (int i = 0; i < a.size(); ++i) {
        if(isOutlier(a[i],percentil)){                              //k
            outliers.push_back(i);                                  //k_{1}
        }
    }
}
```

$\#O_{setOutliersPositions} : k_0 + (\sum_{i=0}^{|a|-1} k_{for} + k + k_1)$

$\Rightarrow T_{setOutliersPositions}(n) = k_0 + n * (k) = n * k \in \mathcal{O}(n)$

isOutlier

```
bool isOutlier (int &n,int &percentil){
    return n>percentil;                         //k_{0}+k_{1}
}
```

$\#O_{isOutlier} : k_0 + k_1$

$\Rightarrow T_{isOutlier} = k \in \mathcal{O}(1)$

modifyOnlyOutliers

```
void modifyOnlyOutliers (audio &a, vector<int> &outliers, int &percentil){
    for (int i = 0; i < a.size() ; ++i) {
```

```
        if(isOutlier(a[i],percentil)){                          //k
            modifyOutlier(a,outliers,percentil,i);              //k*n
        }
    }
}
```

$\#O_{modifyOnlyOutliers} : (\sum_{i=o}^{|a|-1} k + \#O_{modifyOutlier})$

$\Rightarrow T_{modifyOnlyOutliers}(n) = (\sum_{i=o}^{n-1} k_{for} + k + k * n)$

$\Rightarrow T_{modifyOnlyOutliers}(n) = n * (k + k * n) + k * n^2 \in \mathcal{O}(n^2)$

La condición del if, se va a cumplir como mucho outliers veces, y claramente la cantidad de outliers depende de n, pues $0 \leq |\text{outliers}| < \frac{5}{100} * n$. Por lo tanto, en el peor caso, entra al if k*n veces.

modifyOutlier

```
void modifyOutlier(audio &a, vector<int> &outliers, int &percentil,int &i){
    tuple<bool,int> noOutlierRightInfo = noOutlierRight(a,i,percentil);         //k_{0} + k*n
    tuple<bool,int> noOutlierLeftInfo = noOutlierLeft(a,i,percentil);           //k_{1} + k*n

    bool haveOutlierRight = get<0>(noOutlierRightInfo);                         //k
    bool haveOutlierLeft = get<0>(noOutlierLeftInfo);                           //k
    int positionOfFirstNoOutlierRight= get<1>(noOutlierRightInfo);              //k
    int positionOfFirstNoOutlierLeft= get<1>(noOutlierLeftInfo);                //k

    if(haveOutlierRight && haveOutlierLeft){                                    //k
        int sumOfBoth = a[positionOfFirstNoOutlierRight] + a[positionOfFirstNoOutlierLeft];//k
    } else if(haveOutlierRight){                                                //k
        a[i]= a[positionOfFirstNoOutlierRight];                                 //k
    } else if(haveOutlierLeft){                                                 //k
        a[i]= a[positionOfFirstNoOutlierLeft];                                  //k
    }
}
```

$\#O_{modifyOutlier} : k_0 + \#O_{noOutlierRight} + k_1 + \#O_{noOutlierRight} + k_{deTooodasLasKsiguientes})$

$\Rightarrow T_{modifyOutlier}(n) = k_0 + k * n + k_1 + k * n + k$

$\Rightarrow T_{modifyOutlier}(n) = k + k * n = k * n \in \mathcal{O}(n)$

noOutlierRight

```
tuple<bool,int> noOutlierRight(audio &a,int &outlierPosition,int &percentil){
    for (int i = outlierPosition + 1 ; i < a.size() ; ++i) {
        if (!isOutlier(a[i],percentil)){                                //k
            return make_tuple(true,i);                                  //k_{0}
```

```
        }
    }
    return make_tuple(false,0);                                    //k_{1}
}
```

$\#O_{noOutlierRight} : (\sum_{i=outlierPosition+1}^{|a|-1} + k_{for} + k_{not} + \#O_{isOutlier} + k_0) + k_1$

$\Rightarrow T_{noOutlierRight}(n) = (\sum_{i=outlierPosition+1}^{n-1} + k) + k$

$\Rightarrow T_{noOutlierRight}(n) = n * k + k = n * k \in \mathcal{O}(n)$

noOutlierLeft es practicamente igual, por lo tanto tambien es $\mathcal{O}(n)$

maximosTemporales

```
void maximosTemporales(audio a, int profundidad, vector<int> tiempos, vector<int>& maximos,
vector<pair<int,int> >& intervalos) {
    maximos ={};                                                   //k_{1}
    intervalos = {};                                               //k_{2}
    int maxNum = 0;                                                //k_{3}
    int ultimo=0;                                                  //k_{4}
    for (int i = 0; i < tiempos.size() ; ++i) {                    //sumatoria y k_{5}
        for (int j = 0; j < a.size(); ++j) {                       //sumatoria y k_{6}
            if(abs(a[j])>abs(maxNum)){                             //k_{7}
                maxNum=a[j];                                       //k_{8}
            }
            if((j+1)%tiempos[i]==0){                               //k_{9}
                maximos.push_back(maxNum);                         //k_{10}
                maxNum=0;                                          //k_{11}
                intervalos.push_back(make_pair(j+1-tiempos[i],j)); //k_{12}
                ultimo=j;                                          //k_{13}
            }else if(j+1==a.size() && (j+1)%tiempos[i]!=0){         //k_{14}
                maximos.push_back(maxNum);                         //k_{15}
                maxNum=0;                                          //k_{16}
                intervalos.push_back(make_pair((ultimo+1),ultimo+tiempos[i]));  //k_{17}
            }
        }
    }
}
```

$\#O_{maximosTemporales} : k_1+k_2+k_3+k_4+(\sum_{i=0}^{|tiempos|-1} k_5+(\sum_{i=0}^{|a|-1} k_6+k_7+k_8+k_9+k_{10}+k_{11}+k_{12}+k_{13}+k_{14}+k_{15}+k_{16}+k_{17}))$

$\Rightarrow T_{maximosTemporales} = k_1 + k_2 + (tiempos * (k_5 + (a * (k_6 + k_7 + k_8 + k_9 + k_{10} + k_{11} + k_{12} + k_{13} + k_{14} + k_{15} + k_{16} + k_{17}))))$

$\Rightarrow T_{maximosTemporales}(n, m) = k_1 + k_2 + (m * (k_5 + (n * (k_6 + k_7 + k_8 + k_9 + k_{10} + k_{11} + k_{12} + k_{13} + k_{14} + k_{15} + k_{16} + k_{17}))))$

(m siendo la cantidad/longitud de tiempos)

$\Rightarrow T_{maximosTemporales}(n, m) = k'_1 + (m * (k'_2 + (n * k'_3)))$

$\Rightarrow T_{maximosTemporales}(n,m) = k'_1 + m * k'_2 + m * n * k'_3$

$\Rightarrow T_{maximosTemporales}(n,m) = m + m * n = m * (n+1) = m * n \in \mathcal{O}(n * m)$ (a partir de 2 m es menor que n siempre)

Ejercicio 2.2

magitudAbsolutaMaxima

```
1  void magnitudAbsolutaMaxima(audio a, int canal, int profundidad, vector<int> &maximos,
2  vector<int> &posicionesMaximos) {
3      maximos={};                                                    //k_{1}
4      posicionesMaximos={};                                          //k_{2}
5      for (int c = 0; c < canal; ++c) {                              //sumatoria y k_{3}
6          int maxCandidate = a[c];                                   //k_{4}
7          int posCandidate = c;                                      //k_{5}
8          for (int i = 0; i < a.size()/canal ; ++i) {                //sumatoria y k_{6}
9              if (abs(a[c + canal*i]) >= abs(maxCandidate)){         //k_{7}
10                 maxCandidate=a[c+canal*i];                         //k_{8}
11                 posCandidate= c+canal*i;                           //k_{9}
12             }
13         }
14         maximos.push_back(maxCandidate);                           //k_{10}
15         posicionesMaximos.push_back(posCandidate);                 //k_{11}
16     }
17 }
```

$\#O_{magitudAbsolutaMaxima} : k_1 + k_2 + (\sum_{i=0}^{canal-1} k_3 + k_4 + k_5 + (\sum_{i=0}^{\frac{|a|}{canal}-1} k_6 + k_7 + k_8 + k_9) + k_{10} + k_{11})$

$\Rightarrow T_{magitudAbsolutaMaxima} = k_1 + k_2 + ((canal) * (k_3 + k_4 + k_5 + (\frac{|a|}{canal}) * (k_6 + k_7 + k_8 + k_9) + k_{10} + k_{11})$

$\Rightarrow T_{magitudAbsolutaMaxima}(n) = k_1 + k_2 + ((canal) * (k_3 + k_4 + k_5 + (\frac{n}{canal}) * (k_6 + k_7 + k_8 + k_9) + k_{10} + k_{11})$

$\Rightarrow T_{magitudAbsolutaMaxima}(n) = k'_1 + ((canal) * (k'_2 + (\frac{n}{canal}) * (k'_3) + k'_4)$

$\Rightarrow T_{magitudAbsolutaMaxima}(n) = k'_1 + ((canal) * (k'_5 + (\frac{n}{canal}) * (k'_3))$

$\Rightarrow T_{magitudAbsolutaMaxima}(n) = k'_1 + (canal) * k'_5 + (canal)(*\frac{n}{canal}) * k'_3$

$\Rightarrow T_{magitudAbsolutaMaxima}(n) = k'_1 + (canal) * k'_5 + n * k'_3$

$\Rightarrow T_{magitudAbsolutaMaxima}(n) = canal + n$ (en el peor caso canal = n)

$\Rightarrow T_{magitudAbsolutaMaxima}(n) = n + n = 2n = n \in \mathcal{O}(n)$

audiosSoftYHard

```
void audiosSoftYHard(vector<audio> as, int profundidad, int longitud, int umbral,
vector<audio>& soft, vector<audio>& hard) {
    soft={};                                                    //k_{1}
    hard={};                                                    //k_{2}
    for (int i = 0; i < as.size() ; ++i) {
        if (esSoft(as[i], umbral, longitud)) {                  //k*m
            soft.push_back(as[i]);                              //k
        } else if (!esSoft(as[i], umbral, longitud)) {          //k*m
            hard.push_back(as[i]);                              //k
        }
    }
}
```

$\#O_{audiosSoftYHard} : k_1 + k_2 + (\sum_{i=0}^{|as|-1} k_{for} + \#O_{esSoft} + k)$

$\Rightarrow T_{audiosSoftYHard} = k_1 + k_2 + |as| * (k_{for} + k * |as[i]|)$
(as' es el audio dentro del vector de audios y as es el vector de audios en este caso llamo n a la longitud del vector de audios y m a la longitud del mayor audio dentro del vector) *mas info en auxiliares.cpp

$\Rightarrow T_{audiosSoftYHard}(n, m) = k_1 + k_2 + n * (k_{for} + k * m)$ constantes

$\Rightarrow T_{audiosSoftYHard}(n, m) = k + n * (k * m) = k * n * m \in \mathcal{O}(n * m)$

Auxiliar de audiosSoftYHard

esSoft

```
bool esSoft(audio &a, int &umbral, int &longitud){
    if (a.size()<longitud){                                     // k
        return true; //                                          //k
    }
    int acum=0;                                                 //k
    for (int i = 0; i < a.size() ; ++i) {
        if(a[i]>umbral && acum == longitud){                    // k
            return false;                                       // k
        }
        if(a[i]>umbral){                                        // k
            acum+=1;                                            // k
        }
        if(a[i]<umbral){                                        // k
            acum=0;                                             // k
        }
    }
    return true;                                                //k
}
```

$\#O_{esSoft} : k + (\sum_{i=0}^{|a|-1} k_{for} + k)$

$\Rightarrow T_{esSoft}(m) = k + (\sum_{i=0}^{m-1} k_{for} + k)$

$\Rightarrow T_{esSoft}(m) = k + m * (k_{for} + k) = k + m * k = m \in \mathcal{O}(m)$

reemplazarSubAudio

```
void reemplazarSubAudio(audio& a, audio a1, audio a2, int profundidad) {
    int acum = 0;                                                          // k_{0}
    int endPositionA1=0;                                                   // k_{1}
    int startPositionA1=0;                                                 // k_{2}
    calculateA1EPositions(a,a1,acum,startPositionA1,endPositionA1);        // k*n
    if (acum!=0){                                                          // k_{3}
        audio finalAudioFragment={};                                       // k
        copyAudioFragment(a,endPositionA1+1,a.size()-1,finalAudioFragment); // k*n
        deleteAudioFragment(a,startPositionA1-1);                          // k*n
        concatV(a,a2);                                                     // k*m
        concatV(a,finalAudioFragment);                                     // k*n
    }
}
```

$\#O_{reemplazarSubAudio} : k_0 + k_1 + k_2 + \#O_{calculateA1Position} + k_3 + k_4 + \#O_{copyAudioFragment} + \#O_{deleteAudioFragment} + \#O_{concatV_1} + \#O_{concatV_2}$

$\Rightarrow T_{reemplazarSubAudio}(n, m) = k_0 + k_1 + k_2 + k * n + k_3 + k_4 + k * n + k * n + k * m + k * n$

$\Rightarrow T_{reemplazarSubAudio}(n, m) = k + k * n + k + k * n + k * n + k * m + k * n$

$\Rightarrow T_{reemplazarSubAudio}(n, m) = k * n + k * m \in \mathcal{O}(n + m)$

Donde n representa a la longitud de a, y m a la longitud de a2
$O_{concatV_2}$ es O(n), ya que finalAudioFragment es k*n
En cambio, $O_{concatV_1}$ $esO(m)$, ya que |a2| es independiente de |a|
En todos los casos los vectores los paso por referencia

Auxiliares de reemplazarSubAudio

calculateA1Position

```
void calculateA1Positions(audio &a, audio &a1,int &acum, int &startPositionA1,int &endPositionA
    for (int i = 0; i < a.size() ; ++i) {
        if (a[i] == a1[acum]) {                                    //k
            acum++;                                                //k
            if (acum == 1) {                                       //k
                startPositionA1 = i;                               //k
            }
            if (acum == a1.size()) {                               //k
                endPositionA1 = i;                                 //k
                break;                                             //k
            }
        } else {
```

```
            acum = 0;                                                    //k
            startPositionA1 = 0;                                         //k
        }
    }
}
```

$\#O_{calculateA1Position} : (\sum_{i=0}^{|a|-1} k)$

$\Rightarrow T_{calculateA1Position}(n) = n * k \in \mathcal{O}(n)$

En este caso con k me estoy refiriendo a todas las constantes dentro del for, ya que las condiciones de los if son constantes, las instrucciones o operacion dentro de los if y else también lo son. También en esa bolsa de constantes entraria el costo de inicializar i, comparar la condición del del for, y realizar la operación sobre la variable iterativa Y al ciclo lo itera como mucho n veces.

copyAudioFragment

```
void copyAudioFragment(audio &a, int start,int end, audio &copy){
    for (int i = start; i <= end ; ++i) {
        copy.push_back(a[i]);                                    //O(1)
    }
}
```

$\#O_{copyAudioFragment} : (\sum_{start}^{end-1} k)$

$\Rightarrow T_{copyAudioFragment}(n) = (n * k_0) * k = n * k \in \mathcal{O}(n)$

Las iteraciones son n * k_0 ya que tanto start y end son posiciones de a, y en el fondo lo que estoy haciendo es iterar una subsequencia de n, por ende itero en función de n. Y en cada iteración $\text{push}_b ackearcuestak$

deleteAudioFragment

```
void deleteAudioFragment(audio &a, int end){
    for (int i = a.size()-1; i > end ; --i) {
        a.pop_back();
    }
}
```

$\#O_{deleteAudioFragment} : (\sum_{end+1}^{|a|-1} k)$

$\Rightarrow T_{deleteAudioFragment}(n) = (n * k_0) * k = n * k \in \mathcal{O}(n)$

Las iteraciones son n * k_0 ya que end es una posicion de a, y en el fondo lo que estoy haciendo es iterar una subsequencia entre end y n, por lo tanto, estoy iterando en función de n. Y en cada iteración $\text{pop}_b ackearcuestak, yaquepop_b ackesO(1)$

copyAudioFragment

```
void copyAudioFragment(audio &a, int &start,int &end, audio &copy){
    for (int i = start; i <= end ; ++i) {
        copy.push_back(a[i]);                              //O(1)
    }
}
```

$\#O_{copyAudioFragment} : (\sum_{start}^{end-1} k)$

$\Rightarrow T_{copyAudioFragment}(n) = (n * k_0) * k = n * k \in \mathcal{O}(n)$

Las iteraciones son n * k_0 ya que tanto start y end son posiciones de a, y en el fondo lo que estoy haciendo es iterar una subsequencia de n, por ende itero en función de n. Y en cada iteración pushbackear cuesta k

concatV

```
void concatV (vector<int>& a, vector<int>& b){
    for (int i = 0; i < b.size(); ++i) {
        a.push_back(b[i]);                              //O(1)
    }
```

$\#O_{deleteAudioFragment} : (\sum_{0}^{|b|-1} k)$

$\Rightarrow T_{deleteAudioFragment} = |b|) * k =$

$\Rightarrow T_{deleteAudioFragment}(m) = m) * k = n * k \in \mathcal{O}(n)$

En este caso m lo denotamos para referirnos a la longitud de b, o sea del segundo vector que toma la funcion como parametro. En k estoy embolsando el costo constante de $\text{push}_b ackear, dettomarb[i], yloscostosconstantespropiosdelfor$

Ejercicio 3 y 4

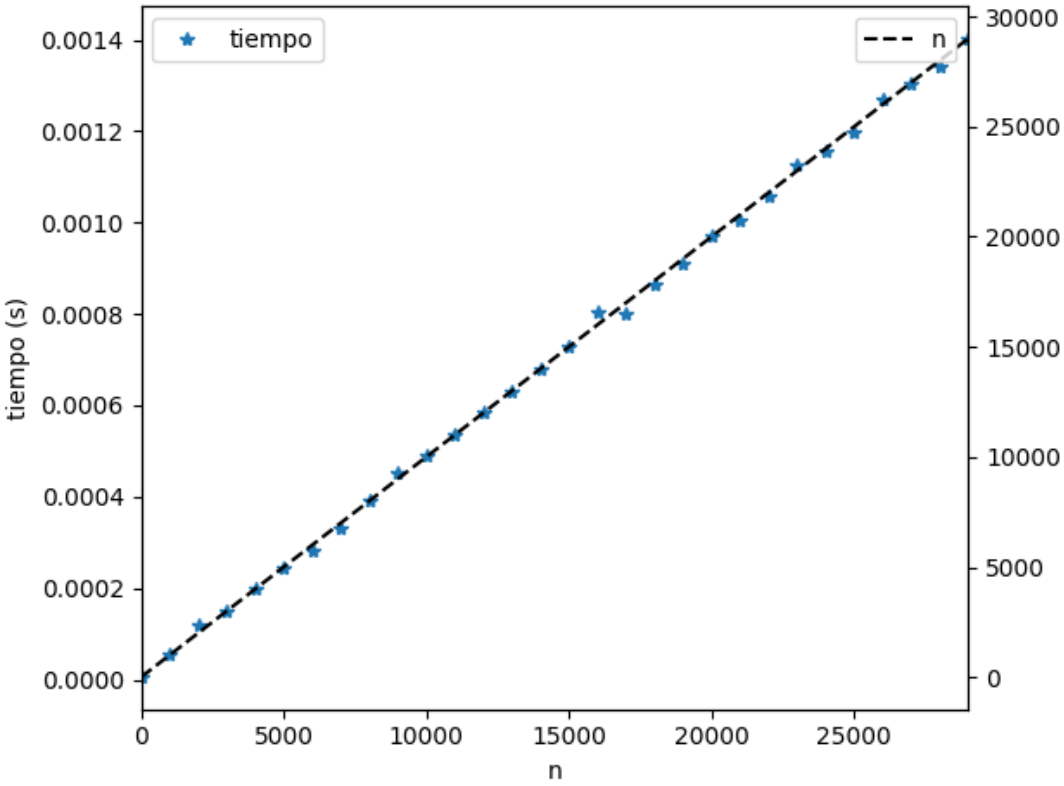
Ambos estan en soluciones.cpp y con sus funciones auxiliares en auxiliares.cpp

Ejercicio 6

El analisis de cobertura se encuentra completo en tpi2atrmasmas/TPI /MIR/cmake-build-debug/CMakeFiles
Con 100 % en soluciones.cpp y 100 % en todas las funciones auxiliares de auxiliares.cpp que se usan para resolver los ejercicios de soluciones.cpp

Ejercicio 2.3

revertirAudio



limpiarAudio