

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SUL-RIO-GRANDENSE - CÂMPUS PASSO FUNDO  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**A CANÇÃO DE VECNA**

Construção de um algoritmo de ordenação adaptado ao contexto de Stranger Things

**CAMILA FLORÃO BARCELLOS**

**PASSO FUNDO - RS**

**2022**

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SUL-RIO-GRANDENSE - CÂMPUS PASSO FUNDO  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**A CANÇÃO DE VECNA**

Construção de um algoritmo de ordenação adaptado ao contexto de Stranger Things

**CAMILA FLORÃO BARCELLOS**

Trabalho avaliativo da primeira etapa  
do semestre letivo 2022/2 da  
disciplina Estrutura de Dados III.

**Professor:** Élder F. F. Bernardi

## RESUMO

O presente trabalho objetiva a elaboração e implementação em linguagem de programação de alto nível de um algoritmo de ordenação linear que apresente a menor complexidade de tempo possível, a fim de garantir uma execução eficiente em curto tempo. O algoritmo será aplicado ao contexto fictício apresentado pela quinta temporada da série televisiva “*Stranger Things*”, em que o vilão Vecna ameaça os habitantes da cidade Hawkins e, para ser derrotado, deve-se decifrar a música correta embaralhada em arquivos de notas musicais.

**Palavras-chave:** algoritmo; ordenação; Vecna; música.

## SUMÁRIO

<b>1 INTRODUÇÃO E CONTEXTUALIZAÇÃO DO CENÁRIO</b>	<b>4</b>
<b>2 CARACTERIZAÇÃO DO PROBLEMA</b>	<b>4</b>
<b>3 INTRODUÇÃO TÉCNICA PARA COMPREENSÃO DO PROBLEMA</b>	<b>4</b>
<b>4 APRESENTAÇÃO DA SOLUÇÃO PROPOSTA</b>	<b>5</b>
4.1. DESCRIÇÃO DA IDEIA GERAL E TÉCNICAS UTILIZADAS	5
4.2. CONSTRUÇÃO E EXPLICAÇÃO DOS ALGORITMOS	5
4.2.1. CARREGAMENTO DO ARQUIVO DE ENTRADA	6
4.2.2. SEPARAÇÃO DOS ARQUIVOS POR MÚSICA	7
4.2.3. CARREGAMENTO RECICLÁVEL DE CADA MÚSICA	8
4.2.4. ORDENAÇÃO DAS MÚSICAS PELA CHAVE ORDEM (COUNTING SORT)	9
4.2.5. ESCRITA DAS NOTAS DAS MÚSICAS EM UM ARQUIVO ABC	11
4.2.6. CONVERSÃO DAS MÚSICA PARA O FORMATO MIDI	12
<b>5 TESTES DE DESEMPENHO DA SOLUÇÃO</b>	<b>12</b>
5.1. DESCRIÇÃO DA IMPLEMENTAÇÃO DO AMBIENTE DE TESTE	12
5.1.1. IMPLEMENTAÇÃO DA SOLUÇÃO	13
5.1.2. METODOLOGIA DE TESTES	13
5.2. TEMPO DE EXECUÇÃO DE CADA SOLUÇÃO PARA OS CASOS	13
5.2.1. MELHOR, MÉDIO E PIOR CASOS	13
5.2.2. ENTRADAS PEQUENAS, MÉDIAS E GRANDES	14
5.2.3. ALGORITMO COUNTING E HEAP SORT	14
5.3. ANÁLISE DE COMPLEXIDADE	14
<b>6 OTIMIZAÇÕES POSSÍVEIS</b>	<b>15</b>
<b>7 CONCLUSÃO E REPRODUÇÃO DA MÚSICA CORRETA</b>	<b>15</b>
<b>REFERÊNCIAS</b>	<b>16</b>

## 1 INTRODUÇÃO E CONTEXTUALIZAÇÃO DO CENÁRIO

A cidade de Hawkins corre perigo com o ataque do vilão Vecna, responsável por eventos paranormais e malignos recorrentes na cidade. Ao capturar sua vítima, Vecna só pode ser impedido de assassiná-la caso ela se concentre em um pensamento marcante e poderoso – como a sua música favorita. Para deter o vilão, portanto, Eleven e seus amigos encarregam a turma 5M1 de Estrutura de Dados III, do quinto semestre do curso de Ciência da Computação do IFSul, da importante tarefa de adentrar a dimensão *upside-down*, controlada por Vecna, e reproduzir as músicas favoritas de todos da equipe simultaneamente, o que enfraqueceria o vilão.

A missão, entretanto, sofre um imprevisto: ao chegar no mundo invertido, a equipe descobre que as músicas estão embaralhadas, o que torna a sua reprodução em um grave perigo devido à irreconhecível distorção melódica na qual foram transformadas.

## 2 CARACTERIZAÇÃO DO PROBLEMA

As músicas foram embaralhadas de forma a conter treze arquivos, cada um contendo uma música correta escondida entre outras falsas e com as suas linhas misturadas entre si. O presente projeto foi encarregado de decifrar o arquivo da música de número três, denominado “*songs3*”.

O problema apresentado, portanto, é da natureza de ordenação. Para solucioná-lo, deve-se ordenar cada música (arquivo) e a ordem de cada nota dentro do seu respectivo arquivo. Assim, cada música será tocada individualmente e será possível determinar qual é a correta.

Ademais, a atmosfera do mundo invertido demanda a necessidade de uma execução no menor tempo possível da operação de ordenação.

## 3 INTRODUÇÃO TÉCNICA PARA COMPREENSÃO DO PROBLEMA

O arquivo *song3*, alvo da ordenação solicitada, foi encontrado em formato de texto (.txt) em duas formas: com as informações distribuídas em um *array* (*song3JSONvector*) e linha a linha (*song3LineByLine*). Em ambas as versões, os dados estão armazenados em formato de arquivo JSON da seguinte forma:

```
{
    arq: inteiro,
    ordem: inteiro,
    notas: string
}
```

A chave “**arq**” indica o arquivo da música, a “**ordem**” indica a linha em que a nota musical deve estar dentro do arquivo, e a “**notas**” indica as notas musicais que devem constar na linha respectiva. Essas chaves serão utilizadas como parâmetro na divisão do arquivo único inicial em músicas individuais e na organização das notas em suas devidas linhas, que ocorrerão por meio da implementação de um algoritmo de ordenação com complexidade compatível com a delimitação temporal estabelecida no problema.

## 4 APRESENTAÇÃO DA SOLUÇÃO PROPOSTA

### 4.1. DESCRIÇÃO DA IDEIA GERAL E TÉCNICAS UTILIZADAS

A solução proposta para o problema apresentado desenvolvida neste projeto consiste em seis etapas de execução: carregamento do arquivo em um *array* (*list*) de elementos com par chave/valor (*structs* ou *dicts*); separação do *array* em arquivos de texto (.txt) individuais; carregamento de cada música em um *array* temporário; ordenação dos *arrays* com base na chave “**ordem**” (linha); escrita apenas da chave “**notas**” de cada *array* em um arquivo de música (.abc); conversão dos arquivos com extensão ABC para o formato tocável MIDI. O arquivo selecionado foi a versão distribuída por linhas devido à utilização de uma leitura linha a linha do arquivo no programa.

Para a implementação, a tecnologia utilizada foi a linguagem de programação Python na versão 3.10 com a importação das bibliotecas “json” e “time” – responsáveis, respectivamente, pela leitura dos dados do arquivo de entrada e pela captura e cálculo do tempo de execução do programa. Cabe ressaltar que a escolha da linguagem de programação foi arbitrária e pautada na facilidade de codificação e manutenção do código.

### 4.2. CONSTRUÇÃO E EXPLICAÇÃO DOS ALGORITMOS

#### 4.2.1. CARREGAMENTO DO ARQUIVO DE ENTRADA

A execução do método principal (*main*) do programa inicia com o armazenamento do tempo de início, por meio da biblioteca *time*, e o armazenamento do conteúdo dos dados das músicas em um *array* por meio de uma função de carregamento do arquivo de texto armazenado na pasta “entrada” do projeto.

```
# Captura o tempo de início da execução
inicio = time.time()

# Carrega as informações do arquivo txt de entrada para um vetor
vet_dados = carregar_arquivo('./entrada/songs3LineByLine.txt')
```

*Printscreen da captura de tempo e chamada da função*

O método **carregar\_arquivo()** possui o parâmetro **nome** que indica qual o arquivo alvo para carregar ao *array*. A função utiliza o método **ler\_linha()** para percorrer cada linha do arquivo e armazená-la como um dado de *dict* individual no *array* que será retornado. A biblioteca *json* possibilita a leitura do arquivo.

```
# Função de leitura do arquivo txt linha a linha
def ler_linha(nome):
    return [json.loads(line) for line in open(nome, 'rb')]

# Função de carregamento do conteúdo do txt para um vetor
def carregar_arquivo(nome):
    vet = []
    for i in ler_linha(nome):
        dado = {
            'arq': i['arq'],
            'ordem': i['ordem'],
            'notas': i['notas']
        }
        vet.append(dado)
    return vet
```

*Printscreen dos métodos de carregamento e leitura*

#### 4.2.2. SEPARAÇÃO DOS ARQUIVOS POR MÚSICA

Após o carregamento do arquivo para o *array* **vet\_dados**, este irá conter todas as músicas embaralhadas em si. Para possibilitar a ordenação delas, o *array* **vet\_dados** foi separado em arquivos de texto (.txt) diferentes para cada música que, posteriormente, serão utilizados para ordenar as notas em suas devidas linhas por meio de um algoritmo de ordenação.

```
# Separa as músicas em arquivos próprios
separar_arquivos(vet_dados)
```

*Printscreen da chamada da função sobre o array vet\_dados*

O método **separar\_arquivos()** possui o parâmetro **vet** que indica qual o *array* será alvo da separação – neste caso, o **vet\_dados**. A função percorre todos os elementos do *array* montando um **nome** para o arquivo que será gerado, este seguindo o modelo “**arq\_{chave arq}.txt**”. O nome gerado será utilizado para abrir o arquivo com o método **open()** na pasta “temp” localizada na pasta “saida” do projeto. O arquivo é aberto com o parâmetro **a**, indicando um modo de escrita que insere os novos dados no final do arquivo – este que, caso não exista, será criado.

A escrita ocorre com o método **write()** que escreve o conteúdo do elemento do *array*, substituindo aspas simples por duplas e adicionando uma quebra de linha ao final, a fim de manter a estrutura linha a linha do arquivo original.

```
# Função que reúne as infos de cada arquivo em um txt
def separar_arquivos(vet):
    for i in vet:
        nome = 'arq_' + str(i.get('arq')) + '.txt'
        arquivo = open('./saida/temp/' + nome, 'a')
        arquivo.write(str(i).replace("'", '"') + '\n')
```

*Printscreen do método de separação de arquivos*



#### 4.2.3. CARREGAMENTO RECICLÁVEL DE CADA MÚSICA

Com as músicas armazenadas em seus próprios arquivos individuais, torna-se possível realizar a ordenação das suas notas musicais.

Para criar uma implementação reciclável de memória, utilizou-se um laço que irá percorrer todas as músicas e armazená-las em um *array* que será constantemente atualizado. O percurso no laço acontece com o auxílio dos métodos **menor\_valor()** e **maior\_valor()**, que encontram esses respectivos elementos com base na chave “**arq**” e determinam a faixa de incremento do laço. Como os arquivos não possuem chaves “**arq**” sequenciais, o método **existe()** garante que iremos acessar um arquivo válido para armazenamento e posterior ordenação.

Ao encontrar o índice de uma música existente, grava-se com ele o nome do arquivo que, como relatado no tópico anterior, está localizado no diretório “**saida/temp**” do projeto. Utiliza-se novamente, então, o método **carregar\_arquivo()**.

```
# Carrega as informações de cada arquivo em um vetor temporário,
# ordena com Counting Sort e gera arquivo de música final
for i in range(menor_valor(vet_dados, 'arq'), maior_valor(vet_dados, 'arq') + 1):
    if existe(str(i), vet_dados):
        nome = './saida/temp/arq_' + str(i) + '.txt'
        vet_temp = carregar_arquivo(nome)
        vet_ordenado = counting_sort(vet_temp, maior_valor(vet_temp, 'ordem'))
        reescrever_arquivos(vet_ordenado, str(i))
```

**Printscreen da reutilização de um *array* no percurso pelas músicas**

Abaixo, seguem as funções auxiliares utilizadas para verificação e validação:

```
# Função que encontra o menor valor de um determinado tipo no vetor
def menor_valor(vet, tipo):
    menor = vet[0].get(tipo)
    for i in vet:
        if i.get(tipo) < menor:
            menor = i.get(tipo)
    return menor
```

```

# Função que encontra o maior valor de um determinado tipo no vetor
def maior_valor(vet, tipo):
    maior = vet[0].get(tipo)
    for i in vet:
        if i.get(tipo) > maior:
            maior = i.get(tipo)
    return maior

# Função que verifica a existência no vetor
def existe(arq, vet):
    for i in vet:
        if str(i.get('arq')) == arq:
            return True
    return False

```

#### *Printscreen dos métodos de verificação*

#### 4.2.4. ORDENAÇÃO DAS MÚSICAS PELA CHAVE ORDEM (*COUNTING SORT*)

A ordenação das músicas ocorre por meio do algoritmo de ordenação **Counting Sort**. Ele é um algoritmo de ordenação em tempo linear pautado no conceito de ordenação por contagem, que pressupõe que “cada um dos  $n$  elementos de entrada é um inteiro no intervalo de um a  $k$ , para algum inteiro  $k$ ” (CORMEN et al., 2012). A sua complexidade será analisada no tópico 5.3 com base no código implementado em linguagem Python utilizado no projeto.

Cabe ressaltar que uma característica marcante do algoritmo é a ausência de operadores de comparação, pois a ordenação de cada elemento ocorre pela determinação da quantidade de elementos menores que ele existente no conjunto. Isso possibilita a inserção dos elementos de entrada no *array* de saída diretamente na sua devida posição final. Por exemplo, um elemento com quatro outros menores que ele será inserido na quinta posição.

A implementação do algoritmo ocorre em três etapas após a inicialização de um vetor auxiliar e culmina no retorno de um *array* final contendo os elementos do *array* de entrada ordenados com base na chave “ordem”, conforme abaixo:

```

def counting_sort(vet_entrada, maior_valor):
    # Array auxiliar de contagem inicializado em
    # 0s para armazenar a quantidade de ocorrências
    # de cada elemento de vet_entrada
    count_vet_tam = maior_valor + 1
    count_vet = [0] * count_vet_tam

    # Etapa 1 → Percorre o vet_entrada e incrementa
    # a contagem de cada elemento em 1 (equivale ao
    # índice do elemento no vet_saida)
    for el in vet_entrada:
        count_vet[el.get('ordem')] += 1

    # Etapa 2 → Para cada elemento no count_vet, soma
    # o seu valor com o valor do elemento anterior a ele
    # e armazena como o valor do elemento atual
    for i in range(1, count_vet_tam):
        count_vet[i] += count_vet[i - 1]

    # Etapa 3 → Calcula a posição do elemento com base
    # nos valores contidos em count_vet
    vet_saida = [0] * len(vet_entrada) # encontra o valor do el atual
    i = len(vet_entrada) - 1 # subtrai 1 do valor
    while i ≥ 0: # realiza a operação com todos os elementos
        pos_atual = vet_entrada[i]
        count_vet[pos_atual.get('ordem')] -= 1
        pos_nova = count_vet[pos_atual.get('ordem')]
        vet_saida[pos_nova] = pos_atual
        i -= 1

    return vet_saida # contém os elementos de vet_entrada ordenados

```

**Printscreen do método de ordenação com Counting Sort**

#### 4.2.5. ESCRITA DAS NOTAS DAS MÚSICAS EM UM ARQUIVO ABC

Com as músicas ordenadas em seus devidos arquivos, resta escrever as notas musicais, contidas na chave “**notas**”, em um arquivo em formato ABC. A função **reescrever\_arquivos()** possui o parâmetro **vet** que indica qual o *array* será alvo da separação – neste caso, o **vet\_ordenado** –, e o **num\_arq** que indica qual o arquivo. A função percorre todos os elementos do *array* montando um **nome** para o arquivo que será gerado, este seguindo o modelo “**song\_{num\_arq}.txt**”. O nome gerado será utilizado para abrir o arquivo com o método **open()** na pasta “saida” do projeto. O arquivo é aberto com o parâmetro **a**, indicando um modo de escrita que insere os novos dados no final do arquivo – este que, caso não exista, será criado. A escrita ocorre com o método **write()** que escreve apenas conteúdo da chave “**notas**” do elemento do *array* e adiciona uma quebra de linha ao final.

```
# Função que reúne as infos de cada arquivo em um txt
def reescrever_arquivos(vet, num_arq):
    for i in vet:
        nome = 'song_' + num_arq + '.abc'
        arquivo = open('./saida/' + nome, 'a')
        arquivo.write(str(i.get('notas')) + '\n')
```

*Printscreen do método de reescrita de arquivos*

Ao finalizar o percurso no laço e gerar todas músicas devidamente ordenadas, captura-se o tempo final de execução do programa e realiza-se o cálculo para verificar o tempo total de execução arredondado em segundos.


```
# Captura o tempo de fim da execução e exibe o tempo total
# de execução do programa
fim = time.time()
print('> TEMPO DE EXECUÇÃO:', round(fim - inicio))
```

*Printscreen da captura, cálculo e exibição de tempo*

#### 4.2.6. CONVERSÃO DAS MÚSICA PARA O FORMATO MIDI

Após a execução do programa em Python, obtemos quatorze músicas em formato ABC. Para possibilitar a sua reprodução e a identificação da música correta, deve-se converter os arquivos para o formato MIDI.

A conversão foi realizada por meio de um *bash* criado e executado com a seguinte concatenação de comandos na pasta “saida” do projeto:



```

1  #!/bin/bash
2
3  abc2midi song_2.abc -o song_2.mid ; abc2midi song_8.abc -o
   song_8.mid ; abc2midi song_13.abc -o song_13.mid ; abc2midi
   song_14.abc -o song_14.mid ; abc2midi song_22.abc -o
   song_22.mid ; abc2midi song_25.abc -o song_25.mid ; abc2midi
   song_37.abc -o song_37.mid ; abc2midi song_41.abc -o
   song_41.mid ; abc2midi song_44.abc -o song_44.mid ; abc2midi
   song_67.abc -o song_67.mid ; abc2midi song_71.abc -o
   song_71.mid ; abc2midi song_76.abc -o song_76.mid ; abc2midi
   song_82.abc -o song_82.mid ; abc2midi song_94.abc -o
   song_94.mid

```

**Printscreen do terminal Linux com os comandos de conversão**

Após a conversão, obtemos as quatorze músicas em formato MIDI (.mid) na pasta “saida” do projeto. As músicas, agora, podem enfim ser reproduzidas para encontrar a verdadeira.

## 5 TESTES DE DESEMPENHO DA SOLUÇÃO

### 5.1. DESCRIÇÃO DA IMPLEMENTAÇÃO DO AMBIENTE DE TESTE

Os ambientes de teste da solução proposta foram duas IDEs comumente utilizadas para o desenvolvimento de códigos em linguagem Python: **Visual Studio Code** (Microsoft) e **PyCharm** (JetBrains). Em ambas as IDEs, utilizou-se a biblioteca *time* para capturar o tempo total de execução do algoritmo *Counting Sort* no melhor, médio e pior caso de ordenação.

O programa foi executado em um notebook com processador Intel Core i7 de sexta geração, 8GB de memória RAM e sistema operacional Linux Mint 21.

### 5.1.1. IMPLEMENTAÇÃO DA SOLUÇÃO

A realização dos testes foi composta pela leitura do arquivo “*arq\_2.txt*” para um *array* e pela sua ordenação com o algoritmo *Counting Sort*. Para o teste de tamanho de entradas, contabilizou-se apenas o de ordenação, enquanto para o teste de casos foi contabilizado todo o tempo de execução devido à necessidade de separação dos arquivos.

### 5.1.2. METODOLOGIA DE TESTES

A escolha dos arquivos para teste da solução de deu por representarem os casos em que nenhuma mudança seria necessária (melhor caso), devido ao arquivo já estar ordenado; em que existe uma desordem sem padrão (médio caso); em que todos os elementos devem ser modificados pois estão em ordem inversa (pior caso).

Para os testes de tamanho de entrada, avaliou-se o desempenho da aplicação com o arquivo “*songs3LineByLine.txt*” como caso de entrada grande (cerca de 300 mil linhas), um arquivo modificado com cerca de 50 mil linhas para o caso de entrada média e, para pequena, um arquivo contendo 100 linhas.

## 5.2. TEMPO DE EXECUÇÃO DE CADA SOLUÇÃO PARA OS CASOS

O cálculo do tempo de execução foi efetuado pela diferença entre o tempo final e inicial do programa, capturados por meio da biblioteca *time* e arredondados em segundos.

### 5.2.1. MELHOR, MÉDIO E PIOR CASOS


	Melhor caso	Médio caso	Pior caso
<b>VSCode</b>	0.033s	0.016s	0.030s
<b>PyCharm</b>	0.028s	0.013s	0.020s

### 5.2.2. ENTRADAS PEQUENAS, MÉDIAS E GRANDES

	Pequenas	Médias	Grandes
<b>VSCode</b>	0s	5s	26s
<b>PyCharm</b>	0s	5s	20s

### 5.2.3. ALGORITMO COUNTING E HEAP SORT

CountingSort	HeapSort
0.46s	1.17s



```
Run: TesteAlgoritmos x
/usr/bin/python3.10 /home/camila/Dev/Python/ED3_2022_2/Testes/TesteAlgoritmos.py
Tempo CS: 0.4656403064727783
Tempo HPS: 1.1764583587646484
Process finished with exit code 0
```

*Printscreen do console após a execução do teste de algoritmos*

## 5.3. ANÁLISE DE COMPLEXIDADE

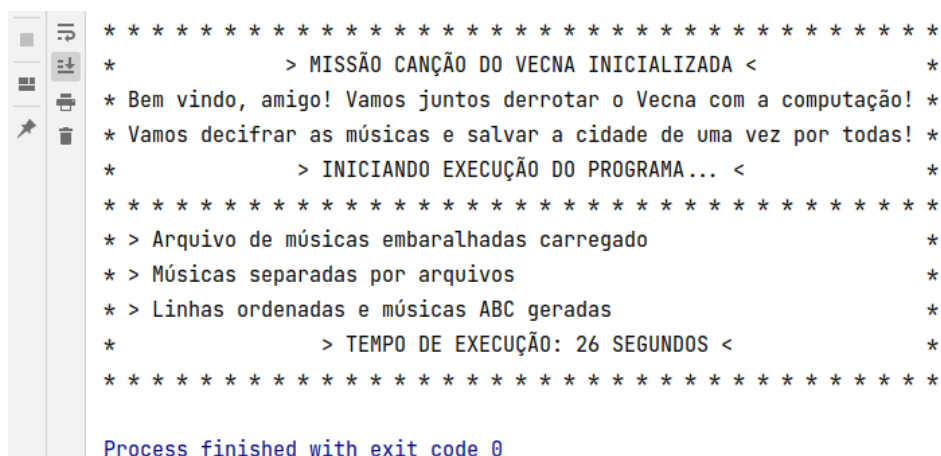
Conforme apresentado no tópico 4.2.3, o algoritmo **Counting Sort** possui uma complexidade analisada sobre os parâmetros  $n$  (tamanho do *array* de entrada) e  $k$  (maior valor do *array*). A primeira etapa itera sobre o *array* de entrada  $n$  vezes para inicializar o *array* de contagem, a complexidade é de  $O(n)$ . A segunda etapa itera sobre o *array* de contagem  $k$  vezes para calcular a quantidade de ocorrências do elemento, logo a complexidade é de  $O(k)$ . A terceira etapa, por sua vez, que realiza a ordenação de fato, itera em um *loop* de *while* sobre  $n$ , então a complexidade é de  $O(n)$ . Portanto, analisando o conjunto de etapas percorridas pelo algoritmo, conclui-se que a sua complexidade de tempo é de  **$O(n+k)$** , o que o configura como um algoritmo de ordenação linear (STAMENIC, 2021).

## 6 OTIMIZAÇÕES POSSÍVEIS

Uma otimização possível à aplicação seria a implementação de vetores para armazenar as músicas separadas, em vez de arquivos de texto. Isso otimizaria o tempo de execução do programa e, conseqüentemente, o desempenho do algoritmo, pois grande parte do tempo disposto na execução ocorre devido às operações de leitura e gravação dos arquivos.

## 7 CONCLUSÃO E REPRODUÇÃO DA MÚSICA CORRETA

A implementação e execução do programa em Python desenvolvido neste trabalho tornou possível a resolução eficiente do problema proposto referente ao contexto fictício do vilão Vecna, que exigia a ordenação em tempo ágil de diversas músicas embaralhadas em um mesmo arquivo.



```

* * * * *
*           > MISSÃO CANÇÃO DO VECNA INICIALIZADA <           *
* Bem vindo, amigo! Vamos juntos derrotar o Vecna com a computação! *
* Vamos decifrar as músicas e salvar a cidade de uma vez por todas! *
*           > INICIANDO EXECUÇÃO DO PROGRAMA... <           *
* * * * *
* > Arquivo de músicas embaralhadas carregado                    *
* > Músicas separadas por arquivos                                *
* > Linhas ordenadas e músicas ABC geradas                        *
*           > TEMPO DE EXECUÇÃO: 26 SEGUNDOS <                  *
* * * * *

Process finished with exit code 0

```

**Printscreen do console após a execução da aplicação**

Após a execução do programa e da conversão dos arquivos ABC em MIDI, geraram-se quatorze músicas. Cada uma delas foi executada na busca da correta e a verdadeira foi encontrada: “song\_2”, equivalente à música “***Under The Bridge***”, da banda **Red Hot Chili Peppers**.



## REFERÊNCIAS

CORMEN, Thomas H; RIVEST, Ronald L.; LEISERSON, Charles E.; STEIN, Clifford. **Algoritmos - Teoria e Prática**. 3. ed. São Paulo-SP, Elsevier, 2012.

STAMENIC, Dimitrije. **Counting Sort in Python**. StackAbuse, 2021. Disponível em: <https://stackabuse.com/counting-sort-in-python/>. Acesso em: 18, out de 2022.

POPOVIC, Olivera. **Heap Sort in Python**. StackAbuse, 2020. Disponível em: <https://stackabuse.com/heap-sort-in-python/>. Acesso em: 23, out de 2022.