Universidade Tecnológica Federal do Paraná
Departamento Acadêmico de Informática
CSF13 – Fundamentos de Programação 1
Prof. Bogdan Tomoyuki Nassu
Profa. Leyza Baldo Dorini

Trabalho 2

Esta é a especificação do segundo trabalho da disciplina CSF13 - Fundamentos de Programação 1, profs. Bogdan T. Nassu e Leyza B. Dorini, para o período 2022/2.

I) Equipes

O trabalho deve ser feito em dupla. Trabalhos individuais só serão aceitos se devidamente justificados e previamente autorizados pelos professores. A justificativa deverá ser um motivo de força maior: questões pessoais como “prefiro trabalhar sozinho” ou “não conheço ninguém da turma” não serão aceitas. Durante as aulas, algumas duplas podem ser convocadas para explicar como está a divisão do trabalho (ver item IV).

A discussão entre colegas para compreender a estrutura do trabalho e esta especificação é recomendada e estimulada, mas cada equipe deve apresentar suas próprias soluções para os problemas propostos. Indícios de fraude (cópia) podem levar à avaliação especial (ver item IV). Não compartilhe códigos (fonte nem pseudocódigos)!!!

II) Entrega

O prazo de entrega é 09/11/2022, 23:59. Trabalhos entregues após esta data terão sua nota final reduzida em 0,00025% para cada segundo de atraso. Cada aluno deve entregar, através da página da disciplina no Classroom, dois arquivos (separados, sem compressão):

- Um arquivo chamado *t2-x-y.c* (*t2-x.c*, caso o trabalho seja feito individualmente), onde *x* e *y* são os números de matrícula dos alunos. O arquivo deve conter as implementações das funções pedidas (com cabeçalhos idênticos aos especificados). As funções pedidas podem fazer uso de outras funções, sejam funções da biblioteca-padrão, funções criadas pelos autores, ou as próprias funções pedidas (i.e. uma função pedida pode invocar outras funções pedidas!). Os autores do arquivo devem estar identificados no início, através de comentários.

IMPORTANTE: as funções pedidas não envolvem interação com usuários. Elas não devem imprimir mensagens (por exemplo, através da função `printf`), nem bloquear a execução enquanto esperam entradas (por exemplo, através da função `scanf`). O arquivo também não deve ter uma função `main`.

- Um arquivo no formato PDF chamado *t2-x-y.pdf* (*t2-x.pdf*, caso o trabalho seja feito individualmente), onde *x* e *y* são os números de matrícula dos alunos. Este arquivo deve conter um relatório breve (em torno de 1 a 2 páginas), descrevendo (a) a contribuição de cada membro da equipe, (b) os desafios encontrados, e (c) a forma como eles foram superados. Não é preciso seguir uma formatação específica. Os autores do arquivo devem estar identificados no início.

III) Avaliação (normal)

Todos os testes serão feitos usando a IDE Code::Blocks. Certifique-se de que o seu trabalho pode ser compilado e executado a partir dela.

Os trabalhos serão avaliados por meio de baterias de testes automatizados. A referência será um conjunto de funções implementadas pelos professores, sem “truques” ou otimizações sofisticadas. Os resultados dos trabalhos serão comparados àqueles obtidos pela referência. Cada função será avaliada individualmente. Os seguintes pontos serão avaliados:

III.a) Compilação. Cada erro que impeça a compilação do arquivo implica em uma redução de 50% no peso de uma função caso o erro esteja no seu corpo, ou uma penalidade de 10 pontos em outros casos. Certifique-se que seu código compila!!!

III.b) Corretude. As funções devem produzir o resultado correto para todas as entradas testadas. Uma função que produza resultados incorretos terá sua nota reduzida. Quando possível, o professor corrigirá o código até que ele produza o resultado correto. A cada erro corrigido, a nota da função será multiplicada por um valor: 0.5 se o erro levar a resultados com diferenças óbvias para os exemplos, 0.7 se o erro implicar em erros como repetições infinitas ou acessos inválidos à memória, ou 0.8 em outros casos.

III.c) Atendimento da especificação. Serão descontados até 10 pontos para cada item que não esteja de acordo com esta especificação, como nomes de arquivos e funções fora do padrão.

III.d) Documentação. Comente a sua solução para cada função. Não é preciso detalhar tudo linha por linha, mas forneça uma descrição geral da sua abordagem, assim como comentários sobre estratégias que não fiquem claras na leitura das linhas individuais do programa. Uma função sem comentários terá sua nota reduzida em até 25%.

III.e) Organização e legibilidade. Use a indentação para tornar seu código legível, e mantenha a estrutura do programa clara. Evite construções como *loops* infinitos terminados somente por `break`, ou *loops* `for` com várias inicializações e incrementos, mas sem corpo. Evite também replicar blocos de programa que poderiam ser melhor descritos por repetições. Um trabalho desorganizado ou cuja legibilidade esteja comprometida terá sua nota reduzida em até 30%.

III.f) Variáveis com nomes significativos. Use nomes significativos para as variáveis – lembre-se que `n` ou `x` podem ser nomes aceitáveis para um parâmetro de entrada que é um número, mas uma variável representando uma “soma total” será muito melhor identificada como `soma_total`, `soma` ou `total`; e não como `t`, `aux2` ou `foo`. Uma função cujas variáveis internas não tenham nomes significativos terá sua nota reduzida em até 20%.

III.g) Desempenho. Se a estrutura lógica de uma função for pouco eficiente, por exemplo, realizando muitas operações desnecessárias ou redundantes, a sua nota pode ser reduzida em até 20%.

IV) Avaliação (especial)

Indícios de fraude ou de divisão desigual do trabalho podem levar a uma avaliação especial, com os alunos sendo convocados e questionados sobre aspectos referentes aos algoritmos usados e à implementação. Além disso, alguns alunos podem ser selecionados para a avaliação especial, mesmo sem indícios de fraude, caso exista uma grande diferença entre a nota do trabalho e a qualidade das soluções apresentadas em atividades anteriores, ou se a explicação sobre o funcionamento de uma função for pouco clara.

V) Nota

A nota do trabalho será igual à soma das notas de cada função, mais a nota do relatório. A descrição de cada função indica a sua nota máxima. A nota máxima para o relatório é de 5 pontos.

VI) Apoio

Os professores estarão disponíveis para tirar dúvidas a respeito do trabalho, nos horários de atendimento previstos (e em casos excepcionais, fora deles). A comunicação por e-mail ou via Discord pode ser usada para pequenas dúvidas sobre aspectos pontuais.

Instruções para "montar" o projeto:

Será disponibilizado um "pacote" contendo os seguintes arquivos:

1) Um arquivo trabalho2.h, contendo as declarações das funções, que deve ser incluído no seu arquivo .c através da diretiva `#include`.

2) Arquivos contendo declarações e funções para a manipulação de arquivos no formato Microsoft Wave (.wav): wavfile.c e wavfile.h. Note que as funções do trabalho NÃO PRECISAM acessar o módulo wavfile!

3) Exemplos de programas para teste e arquivos de áudio demonstrando os resultados produzidos.

No Code::Blocks, para compilar e usar no seu programa as funções presentes no arquivo wavfile.c, crie um projeto e adicione o arquivo ao mesmo. Crie outro arquivo .c para as suas funções. Cada arquivo contendo um exemplo de programa contém uma função `main`, portanto apenas um dos exemplos deve ser incluído no projeto por vez. Recomenda-se também a criação de outros programas para testar as funções.

Sobre as funções do módulo wavfile:

Observação: você <u>não precisa</u> compreender o funcionamento do módulo wavfile, mas conseguir utilizá-lo pode ajudar a realizar testes mais avançados. Em particular, a função `writeSamplesAsText` pode ser útil para depurar o seu programa.

O módulo wavfile implementa o tipo `WavHeader` e um conjunto de funções que podem ser usadas para testar as funções do trabalho. O tipo `WavHeader` é usado para armazenar informações sobre um arquivo no formato Microsoft Wave (.wav). Ele é implementado como uma `struct`, um conceito que ainda não vimos em aula. Por enquanto, apenas trate este tipo como um dos tipos primitivos da linguagem C (`int`, `float`, etc.), seguindo os exemplos. Para usar o tipo `WavHeader` e as funções auxiliares, você deve incluir o arquivo wavfile.h, através da diretiva `#include`, no arquivo .c que usa o tipo ou as funções declaradas.

As funções para manipulação de arquivos trabalham apenas com um sub-conjunto do formato Microsoft Wave. Use-as sempre com áudio não comprimido, no formato PCM, com 16 bits por amostra.

```
int readWavFile (char* filename, WavHeader* header,
                 double** data_l, double** data_r);
```

Abre um arquivo wav e lê o seu conteúdo.

```
Parâmetros:
char* filename: arquivo a ser lido.
WavHeader* header: parâmetro de saída para os dados do cabeçalho.
double** data_l: parâmetro de saída, é um ponteiro para um vetor dinâmico, onde
    manteremos os dados lidos para o canal esquerdo. O vetor será alocado nesta
    função, lembre-se de desalocá-lo quando ele  não for mais necessário! Se
    ocorrerem erros, o vetor NÃO estará alocado quando a função retornar.
double** data_r: igual ao anterior, mas para o canal direito. Se o arquivo for
    mono, o vetor será == NULL.

Valor de Retorno: o número de amostras do arquivo, 0 se ocorrerem erros.
```

```
int writeWavFile (char* filename, WavHeader* header,
                  double* data_l, double* data_r);
```

Escreve os dados de áudio em um arquivo wav. Esta função NÃO testa os dados do cabeçalho, então dados inconsistentes não serão detectados. Cuidado!

```
Parâmetros:
char* filename: nome do arquivo a ser escrito.
WavHeader* header: ponteiro para os dados do cabeçalho.
double* data_l: dados do canal esquerdo.
double* data_r: dados do canal direito. Ignorado se o arquivo for mono.

Valor de Retorno: 1 se não ocorreram erros, 0 do contrário.
```

```
WavHeader generateSignal (int* n_samples, unsigned short num_channels, unsigned
                          long sample_rate, double** data_l, double** data_r);
```

Gera um sinal de conteúdo indeterminado. Use esta função se, em vez de ler os dados de um arquivo, você quiser gerar um sinal.

```
Parâmetros:
int* n_samples: ponteiro para uma variável contendo o número de   amostras a
    gerar para cada canal. Se for maior que o permitido, a função gera um sinal
    menor que o pedido e modifica o valor da variável.
unsigned short num_channels: número de canais a gerar. Precisa ser 1 ou 2,
    qualquer outro valor é automaticamente tratado como 1.
unsigned long sample_rate: taxa de amostragem. Se for == 0, é automaticamente
    ajustada para 44.1kHz.
double** data_l: parâmetro de saída, é um ponteiro para um vetor  dinâmico, onde
    manteremos os dados para o canal esquerdo. O vetor será alocado nesta
    função, lembre-se de desalocá-lo quando ele não for mais necessário!
double** data_r: igual ao anterior, mas para o canal direito. Se num_channels ==
    1, o vetor será == NULL.

Valor de Retorno: o cabeçalho para o sinal gerado.
```

```
void generateRandomData (double* data, int n_samples);
```

Preenche um vetor dado com dados aleatórios (ruído). A função NÃO inicia a semente aleatória.

```
Parâmetros:
double* data: vetor a preencher.
int n_samples: número de amostras no vetor.

Valor de Retorno: NENHUM
```

```
int writeSamplesAsText (char* filename, double* data,
                        unsigned long n_samples);
```

Salva as amostras em um arquivo de texto, uma amostra por linha. Esta função é útil para visualizar os dados e depurar o trabalho.

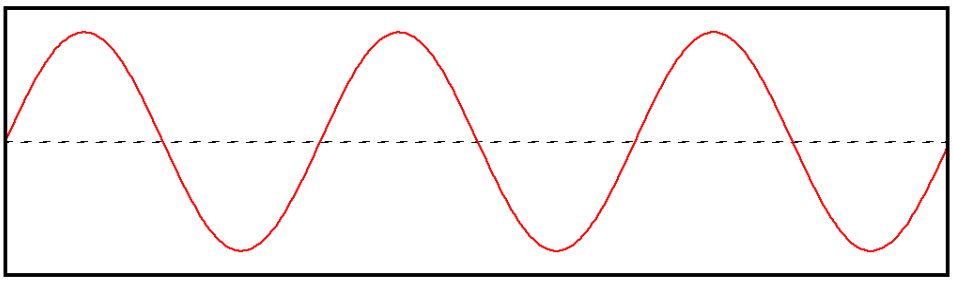
```
Parâmetros:
char* filename: nome do arquivo a ser escrito.
double* data: vetor de dados contendo as amostras.
unsigned long n_samples: número de amostras no vetor.

Valor de Retorno: 1 se não ocorreram erros, 0 do contrário.
```

Função 1 (5 pontos)

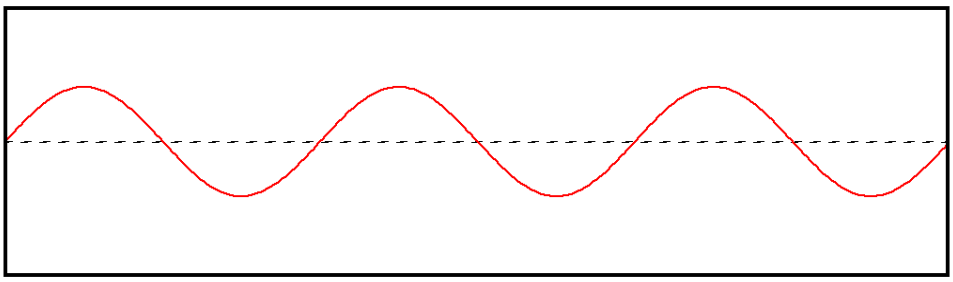
```
void mudaGanho (double* dados, int n_amostras, double ganho);

Parâmetros: double* dados: vetor de dados.
            int n_amostras: número de amostras no vetor.
            double ganho: modificador do ganho.
```

Esta função deve modificar o ganho (volume) de um sinal. Para isso, basta multiplicar cada amostra por um parâmetro `ganho` dado. Se `ganho` = 1, o sinal não é alterado; se `ganho` > 1, o volume do sinal será aumentado; se 0 < `ganho` < 1, o volume do sinal será reduzido; se `ganho` = 0, o sinal será silenciado. Valores negativos para o ganho terão o mesmo efeito, mas invertendo a fase do sinal. Não é preciso tratar de casos nos quais o valor das amostras extrapola o limite máximo permitido pela representação usada (saturação, ou *clipping*). A transformação deve ser feita *in place*, ou seja, o próprio vetor de entrada é usado como saída.

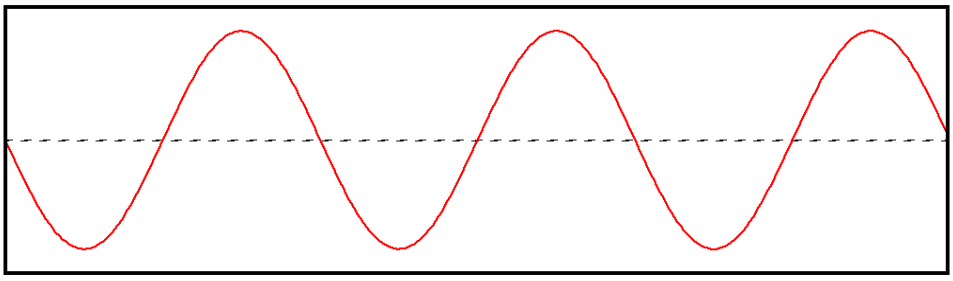
Exemplo:
Sinal original:



Resultado com `ganho` = 0.5:



Resultado com `ganho` = -1. Note que as amostras têm o mesmo valor absoluto, mas com o sinal invertido.

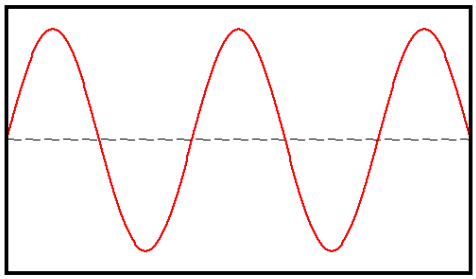


Função 2 (10 pontos)

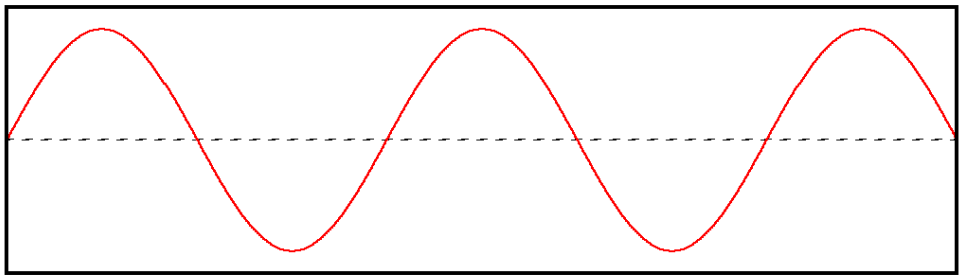
```
void meiaVelocidade (double* dados, int n_amostras, double* saida);

Parâmetros: double* dados: vetor de dados.
            int n_amostras: número de amostras no vetor.
            double* saida: vetor de saída. Deve ter exatamente o dobro do
              tamanho do vetor de entrada.
```

Esta função deve gerar um sinal que é uma versão do sinal de entrada com metade da velocidade. Isso é feito simplesmente copiando cada amostra do sinal de entrada para duas posições consecutivas do vetor de saída. Por exemplo, se as 3 primeiras amostras do sinal de entrada são 0.1, 0.5 e 0.3, as 6 primeiras amostras do sinal de saída serão 0.1, 0.1, 0.5, 0.5, 0.3 e 0.3. Pressuponha que o vetor de saída tem exatamente o dobro do tamanho do vetor de entrada.

Exemplo:
Sinal original:



Sinal de saída:



Função 3 (10 pontos):

```
void dobraVelocidade (double* dados, int n_amostras, double* saida);

Parâmetros: double* dados: vetor de dados.
            int n_amostras: número de amostras no vetor. Deve ser par.
            double* saida: vetor de saída. Deve ter exatamente a metade do
              tamanho do vetor de entrada.
```

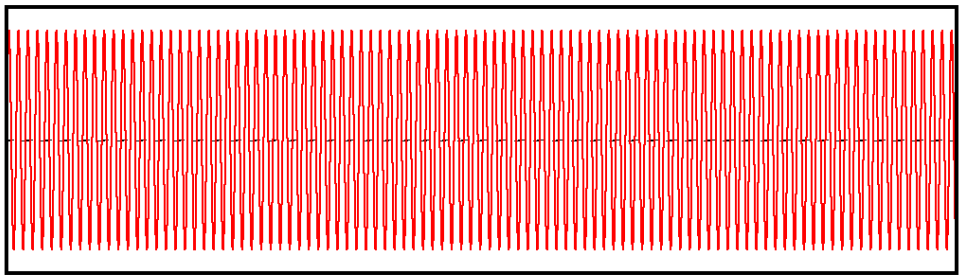
Esta função deve gerar um sinal que é uma versão do sinal de entrada com o dobro da velocidade. Isso é feito simplesmente copiando amostras intercaladas do sinal de entrada para o vetor de saída. Por exemplo, se as 6 primeiras amostras do sinal de entrada são 0.1, 0.2, 0.3, 0.4, 0.5 e 0.6, as 3 primeiras amostras do sinal de saída serão 0.1, 0.3 e 0.5. Pressuponha que o vetor de saída tem exatamente a metade do tamanho do vetor de entrada.

Função 4 (40 pontos):

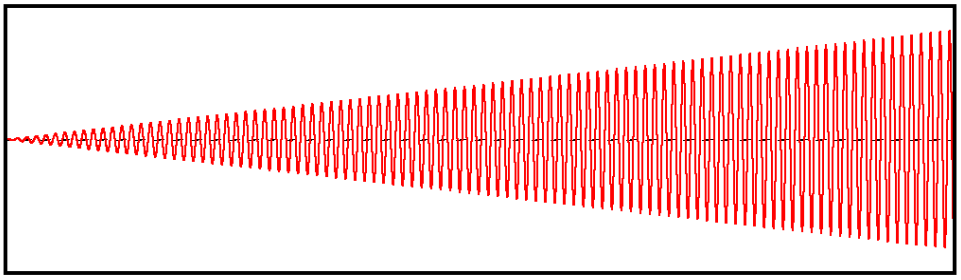
```
void fade (double* dados, int n_amostras, int inicio, int fim, int in_out);

Parâmetros: double* dados: vetor de dados.
            int n_amostras: número de amostras no vetor.
            int inicio: posição de início do fade.
            int fim: posição final do fade. Pode ser maior que n_amostras. Neste
              Neste caso, a variação do ganho funciona como se o vetor fosse
              fosse maior, mas deve parar antes de estourar o tamanho do vetor.
             int in_out: fade in se > 0, fade out se < 0 (ver abaixo).
```

Esta função deve fazer o som "aparecer" ou "desaparecer" em um intervalo dado. Para isso, deve-se aplicar um ganho que varia linearmente de 0 a 1 ou de 1 a 0 dentro de um intervalo. Se `in_out` > 0, `dados[inicio]` terá ganho 0, e `dados[fim]` ganho 1. Se `in_out` < 0, `dados[inicio]` terá ganho 1, e `dados[fim]` ganho 0. Cuidado com os índices do vetor! A transformação deve ser feita *in place*, ou seja, o próprio vetor de entrada é usado como saída.

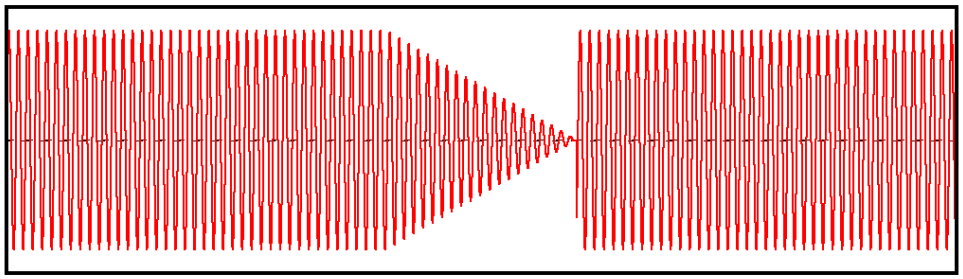
Exemplo:
Um sinal com 10000 amostras:



O sinal após um fade in (`in_out=1`), com `inicio=0` e `fim=9999`:



O sinal após um fade out (`in_out=-1`), com `inicio=2000` e `fim=6000`:



Função 5 (35 pontos):

```
int buscaPadrao (double* dados, int n_amostras, double* padrao,
                 int n_amostras_padrao, double* score);

Parâmetros: double* dados: vetor de dados.
            int n_amostras: número de amostras no vetor.
            double* padrao: padrão procurado.
            int n_amostras_padrao: número de amostras no padrão.
            double* score: parâmetro de saída, serve para armazenar a pontuação
              do melhor casamento encontrado.
```

Técnicas de reconhecimento de padrões são importantes para diversas áreas da computação, como inteligência artificial e visão computacional. Em processamento de áudio, elas são usadas principalmente em aplicações como reconhecimento de voz.

A função `buscaPadrao` deve implementar um esquema simples para reconhecimento de padrões. São dados um sinal de entrada (vetor `dados`, com tamanho `n_amostras`) e um padrão, que também é um sinal (vetor `padrao`, com tamanho `n_amostras_padrao`). O padrão deve ser menor que o sinal de entrada. A função deve retornar a posição inicial do melhor casamento entre o padrão e um trecho do sinal.

Para exemplificar o uso desta função, ouça os arquivos teste.wav e teste5.wav. O arquivo teste5.wav foi gerado pelo programa que está no arquivo teste5_gerar.c. Este programa cria um sinal contendo 5 segundos de ruído com valores entre -0.2 e 0.2, e "cola" o conteúdo do arquivo teste.wav 2 segundos após o início do sinal. Neste caso, se a função procurar pelo padrão teste.wav no arquivo teste5.wav, ela deve retornar a posição 88200, ou alguma posição muito próxima. Você pode testar outras posições para o início do padrão, ou variar o nível de ruído. Evidentemente, neste exemplo estamos procurando pelo padrão em um arquivo gerado por nós mesmos, mas em uma aplicação real, o sinal poderia ter outra origem – por exemplo, em um sistema de reconhecimento de comandos por voz, o sinal seria capturado por um microfone em tempo real, e o padrão estaria associado a algum comando do sistema.

Para localizar um padrão em um arquivo, usaremos uma medida de pontuação baseada na *correlação cruzada*. O padrão é "deslizado" sobre o sinal, alinhando-o a diferentes posições. Para cada alinhamento, calculamos uma pontuação. Sejam *s(x)* a pontuação calculada para uma posição *x*, *f* o sinal de entrada, e *p* o padrão procurado. A pontuação definida pela equação*:

$$s(x) = \frac{\sum_{i=0}^{N} f(x+i) \cdot p(i)}{\sum_{i=0}^{N} p(i)^2}$$

Consideramos que a posição mais provável para o padrão é aquela que tiver a maior pontuação. Esta é a posição que deve ser retornada pela função `buscaPadrao`. Casamentos com pontuação 0.5 ou menor são considerados muito fracos, e descartados. Se não for encontrada pelo menos uma posição com pontuação acima de 0.5, a função deve retornar -1. Se for encontrado mais de um casamento com a pontuação máxima, a função deve retornar a posição do primeiro casamento.

O parâmetro de saída `score` é passado por referência, e deve receber a pontuação do melhor casamento.

Uma otimização simples que pode ser feita nesta função se baseia no fato de que o denominador da equação acima é o mesmo para todas as posições. Com base nisso, é possível realizar o somatório do denominador uma única vez. Na verdade, é possível obter a melhor pontuação fazendo a divisão acima uma única vez, usando apenas o numerador da equação para comparar as posições. Faça esta otimização: a realização de operações redundantes será penalizada.

*A correlação cruzada é o numerador desta equação. Ela não é necessariamente a melhor medida para buscar um padrão, e nem mesmo esta equação é a melhor solução usando correlação cruzada! Mas optamos por deixar assim para que o trabalho não ficasse nem muito simples nem muito complicado.