Este é o CS50

Introdução à Ciência da Computação (CS50)

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

soletrador

Problema a resolver

Para este problema, você implementará um programa que verifica a ortografia de um arquivo, como o mostrado abaixo, usando uma tabela hash.

Demonstração

```
tempore

WORDS MISSPELLED:     30
WORDS IN DICTIONARY:  143091
WORDS IN TEXT:        7573
TIME IN load:         0.05
TIME IN check:        0.01
TIME IN size:         0.00
TIME IN unload:       0.00
TIME IN TOTAL:        0.06

$
```

Recorded with **asciinema**

Código de Distribuição

Para este problema, você deverá estender a funcionalidade do código fornecido pela equipe do CS50.

▼ **Baixe o código de distribuição.**

Faça login em cs50.dev (https://cs50.dev/) , clique na janela do terminal e execute `cd` o comando. Você verá que o prompt do terminal será semelhante ao seguinte:

```
$
```

Em seguida, execute

```
wget https://cdn.cs50.net/2024/fall/psets/5/speller.zip
```

para baixar um arquivo ZIP chamado `speller.zip` para o seu espaço de código.

Em seguida, execute

```
unzip speller.zip
```

para criar uma pasta chamada `speller` . Você não precisa mais do arquivo ZIP, então pode executar

```
rm speller.zip
```

e responda com “y” seguido de Enter quando solicitado para remover o arquivo ZIP que você baixou.

Agora digite

```
cd speller
```

Em seguida, pressione Enter para entrar (ou seja, abrir) esse diretório. Seu prompt agora deve ser semelhante ao abaixo.

```
speller/ $
```

Execute o programa `ls` isoladamente e você deverá ver alguns arquivos e pastas:

```
dictionaries/  dictionary.c  dictionary.h  keys/  Makefile  speller.c  speller50  te
```

Se você encontrar algum problema, siga esses mesmos passos novamente e veja se consegue determinar onde errou!

Fundo

Devido à quantidade de arquivos neste programa, é importante ler esta seção na íntegra antes de começar. Assim, você saberá o que fazer e como fazer!

Teoricamente, para uma entrada de tamanho n , um algoritmo com tempo de execução n é "assintoticamente equivalente", em termos de O , a um algoritmo com tempo de execução $2n$. De fato, ao descrever o tempo de execução de um algoritmo, normalmente nos concentramos no termo dominante (ou seja, o de maior impacto) (neste caso, n *, já que* n poderia ser muito maior que 2). No mundo real, porém, a realidade é que $2n$ parece duas vezes mais lento que n .

O desafio que você tem pela frente é implementar o corretor ortográfico mais rápido possível! Mas, por "mais rápido", estamos falando de tempo real, não de tempo assintótico.

Em `speller.c` , desenvolvemos um programa projetado para verificar a ortografia de um arquivo após carregar um dicionário de palavras do disco para a memória. Esse dicionário, por sua vez, está implementado em um arquivo chamado `dictionary.c` . (Poderia estar implementado em `speller.c` , mas, à medida que os programas se tornam mais complexos, muitas vezes é conveniente dividi-los em vários arquivos.) Os protótipos das funções ali presentes, entretanto, são definidos não em , `dictionary.c` mas em `dictionary.h` . Dessa forma, tanto `speller.c` quanto `dictionary.c` podem acessar `#include` o arquivo. Infelizmente,

não conseguimos implementar a parte de carregamento. Nem a parte de verificação. Deixamos ambas (e um pouco mais) para vocês! Mas primeiro, uma breve apresentação.

Entendimento

`dictionary.h`

Abra o arquivo `dictionary.h` e você verá uma nova sintaxe, incluindo algumas linhas que mencionam `DICTIONARY_H`. Não precisa se preocupar com elas, mas, caso tenha curiosidade, essas linhas apenas garantem que, mesmo que `dictionary.c` e `speller.c` (o que você verá em breve) `#include` este arquivo, `clang` ele só será compilado uma vez.

Observe agora que temos `#include` um arquivo chamado `stdbool.h`. Esse é o arquivo no qual `bool` a própria função está definida. Você não precisou dele antes, pois a Biblioteca CS50 cuidava `#include` disso para você.

Observe também o uso de `__init__` `#define`, uma "diretiva de pré-processador" que define uma "constante" chamada `__init__` `LENGTH` com o valor `true` `45`. É uma constante no sentido de que você não pode (acidentalmente) alterá-la em seu próprio código. Na verdade, `__init__` `clang` substituirá qualquer menção a `__init__` `LENGTH` em seu código por, literalmente, `true` `45`. Em outras palavras, não é uma variável, apenas um truque de localizar e substituir.

Por fim, observe os protótipos de cinco funções: `check`, `hash`, `load`, `size`, e `unload`. Observe como três delas recebem um ponteiro como argumento, conforme o `*`:

```
bool check(const char *word);
unsigned int hash(const char *word);
bool load(const char *dictionary);
```

Lembre-se que `char *` era isso que costumávamos chamar de `string`. Portanto, esses três protótipos são essencialmente apenas:

```
bool check(const string word);
unsigned int hash(const string word);
bool load(const string dictionary);
```

E `const`, enquanto isso, simplesmente afirma que essas strings, quando passadas como argumentos, devem permanecer constantes; você não poderá alterá-las, acidentalmente ou não!

`dictionary.c`

Agora abra `dictionary.c` o arquivo. Observe como, no início do arquivo, definimos um `struct` objeto chamado `node` `node` que representa um nó em uma tabela hash. E declaramos um array de ponteiros global, ` `table` node`, que (em breve) representará a tabela hash que você usará para controlar as palavras no dicionário. O array contém `N` ponteiros para nós e, por

enquanto, definimos `node` `N` como igual a ` `26` null`, para corresponder à `hash` função padrão, conforme descrito abaixo. Você provavelmente desejará aumentar esse valor, dependendo da sua implementação de `node` `hash`.

Em seguida, observe que implementamos `_____ `load`, `_____` `check`, `size`_____` e ` `unload`_____`, mas apenas o suficiente para o código compilar. Observe também que implementamos `_____` `hash` com um algoritmo de exemplo baseado na primeira letra da palavra. Sua tarefa, em última análise, é reimplementar essas funções da maneira mais inteligente possível para que este corretor ortográfico funcione conforme o prometido. E rápido!

`speller.c`

Certo, agora abra o arquivo `speller.c` e dedique algum tempo a analisar o código e os comentários. Você não precisará alterar nada neste arquivo, nem precisa entendê-lo por completo, mas tente ter uma noção de sua funcionalidade. Observe como, por meio de uma função chamada ` `getrusage` benchmark`, faremos um "benchmark" (ou seja, mediremos o tempo de execução) de suas implementações de `__init__`, `__init__` `check`, `load`` `size`__init__` e `__init__` `unload`. Observe também como passamos para `benchmark` `check`, palavra por palavra, o conteúdo de um arquivo para verificação ortográfica. Por fim, relatamos cada erro ortográfico nesse arquivo, juntamente com várias estatísticas.

Note, aliás, que definimos o uso de `speller` como sendo

```
Usage: ./speller [dictionary] text
```

onde `dictionary` se assume que seja um arquivo contendo uma lista de palavras em minúsculas, uma por linha, e `text` seja um arquivo a ser verificado ortograficamente. Como os colchetes sugerem, o fornecimento de `dictionary` é opcional; se este argumento for omitido, `speller` será usado `dictionaries/large` por padrão. Em outras palavras, executando

```
./speller text
```

será equivalente a correr

```
./speller dictionaries/large text
```

Onde `text` está o arquivo que você deseja verificar a ortografia? Basta dizer que a primeira opção é mais fácil de digitar! (Claro, `speller` não será possível carregar nenhum dicionário até que você implemente `load` em `dictionary.c`! Até lá, você verá `Could not load`.)

No dicionário padrão, observe que existem 143.091 palavras, todas as quais precisam ser carregadas na memória! Aliás, dê uma olhada nesse arquivo para ter uma ideia de sua estrutura e tamanho. Note que todas as palavras nesse arquivo aparecem em minúsculas (mesmo, por simplicidade, nomes próprios e acrônimos). De cima para baixo, o arquivo é ordenado lexicograficamente, com apenas uma palavra por linha (cada uma terminando com um ponto

`\n`). Nenhuma palavra tem mais de 45 caracteres e nenhuma palavra aparece mais de uma vez. Durante o desenvolvimento, pode ser útil fornecer `speller` um `dictionary` dicionário próprio com muito menos palavras, para evitar dificuldades na depuração de uma estrutura enorme na memória. O arquivo `.env` contém `dictionaries/small` um desses dicionários. Para usá-lo, execute o comando `npm run dev`.

```
./speller dictionaries/small text
```

Onde `text` está o arquivo que você deseja verificar a ortografia? Não prossiga até ter certeza de que entendeu como `speller` o programa funciona!

É bem provável que você não tenha dedicado tempo suficiente para observar `speller.c`. Volte um quadrado e percorra o mapa novamente!

`texts/`

Para que você possa testar sua implementação `speller`, também fornecemos uma série de textos, incluindo o roteiro de *La La Land* , o texto da Lei de Acesso à Saúde (Affordable Care Act), três milhões de bytes de Tolstói, alguns trechos de *Os Artigos Federalistas* e Shakespeare, e muito mais. Para que você saiba o que esperar, abra e dê uma olhada em cada um desses arquivos, todos localizados em um diretório chamado `texts` dentro do seu `pset5` diretório.

Agora, como você já deve saber por ter lido `speller.c` atentamente, a saída de `speller`, se executado com, digamos,

```
./speller texts/lalaland.txt
```

eventualmente ficará parecido com o exemplo abaixo.

Abaixo, você encontrará alguns exemplos dos resultados. Para fins informativos, selecionamos alguns exemplos de "erros ortográficos". E para não estragar a surpresa, omitimos nossas próprias estatísticas por enquanto.

```
MISSPELLED WORDS

[...]
AHHHHHHHHHHHHHHHHHHHHHHHHHHHHT
[...]
Shangri
[...]
fianc
[...]
Sebastian's
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
```

```
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

`TIME IN load` representa o número de segundos que `speller` o programa gasta executando sua implementação de `load`. `TIME IN check` representa o número `speller` total de segundos que o programa gasta executando sua implementação de `check`. `TIME IN size` representa o número de segundos que `speller` o programa gasta executando sua implementação de `size`. `TIME IN unload` representa o número de segundos que o `speller` programa gasta executando sua implementação de `unload`. `TIME IN TOTAL` é a soma dessas quatro medidas.

Observe que esses tempos podem variar um pouco entre as execuções de `speller`, dependendo do que mais seu espaço de código estiver fazendo, mesmo que você não altere seu código.

Aliás, para que fique claro, por "errado" queremos dizer simplesmente que alguma palavra não está presente no texto `dictionary` fornecido.

`Makefile`

E, por último, lembre-se de que o ` `make` sudo` automatiza a compilação do seu código para que você não precise executá-lo `clang` manualmente com uma série de parâmetros. No entanto, à medida que seus programas crescem, o `sudo` `make` não conseguirá mais inferir pelo contexto como compilar o código; você precisará começar a especificar `make` como compilar seu programa, principalmente quando ele envolver vários `.c` arquivos de origem (ou seja, arquivos .cs), como no caso deste problema. Portanto, utilizaremos um `.cs` `Makefile`, um arquivo de configuração que indica `make` exatamente o que fazer. Abra o arquivo `.cs` `Makefile` e você verá quatro linhas:

1. A primeira linha indica `make` que as linhas subsequentes devem ser executadas sempre que você mesmo executar `make speller` (ou simplesmente `make`).
2. A segunda linha indica `make` como compilar `speller.c` em código de máquina (ou seja, `speller.o`).
3. A terceira linha indica `make` como compilar `dictionary.c` em código de máquina (ou seja, `dictionary.o`).
4. A quarta linha indica `make` para criar um link `speller.o` em `dictionary.o` um arquivo chamado `speller`.

Certifique-se de compilar `speller` executando `make speller` (ou simplesmente `make`). Executar `make dictionary` não funcionará!

Especificação

Muito bem, o desafio agora é implementar, em ordem, `load`, `hash`, `size`, `check`, e `unload` da forma mais eficiente possível usando uma tabela hash, de forma que `TIME IN load`, `TIME IN check`, `TIME IN size`, e `TIME IN unload` sejam minimizados. Certamente, não é óbvio o que significa ser minimizado, visto que esses parâmetros certamente variarão conforme você inserir `speller` valores diferentes para `dictionary` e para `text`. Mas aí reside o desafio, senão a diversão, deste problema. Este problema é a sua chance de projetar. Embora o convidemos a minimizar o espaço, seu maior inimigo é o tempo. Mas antes de começar, algumas especificações da nossa parte.

- Você não pode alterar `speller.c` ou `Makefile`.
- Você pode alterar `dictionary.c` (e, na verdade, deve para completar as implementações de `load`, `hash`, `size`, `check`, e `unload`), mas não pode alterar as declarações (ou seja, protótipos) de `load`, `hash`, `size`, `check`, ou `unload`. Você pode, no entanto, adicionar novas funções e variáveis (locais ou globais) a `dictionary.c`.
- Você pode alterar o valor de `N` `in` `dictionary.c` para que sua tabela hash possa ter mais buckets.
- Você pode alterar `dictionary.h`, mas não pode alterar as declarações de `load`, `hash`, `size`, `check`, ou `unload`.
- Sua implementação `check` deve ser insensível a maiúsculas e minúsculas. Em outras palavras, se ` `foo` is` estiver no dicionário, então ` `check` is` deve retornar verdadeiro, independentemente de sua capitalização; nenhum dos valores `foo`, `foO`, `fOo`, `fOO`, `fOO`, `, `Foo`, `FoO`, `FOo` e `` `FOO` deve ser considerado como erro de ortografia.
- Deixando de lado as maiúsculas e minúsculas, sua implementação de `is` `check` deve retornar apenas `true` palavras que estejam efetivamente em `` `dictionary`. Evite codificar palavras comuns (por exemplo, `is` `the`), para que não passemos à sua implementação um valor `dictionary` sem essas mesmas palavras. Além disso, os únicos possessivos permitidos são aqueles que realmente estão em `` `dictionary`. Em outras palavras, mesmo que ` `foo` is` esteja em `` `dictionary`, `check` `is` deve retornar `false` o valor fornecido `foo's` se `foo's` `is` não estiver também em `` `dictionary`.
- Você pode assumir que qualquer `dictionary` texto passado para o seu programa será estruturado exatamente como o nosso, em ordem alfabética de cima para baixo, com uma palavra por linha, cada uma terminando com um ponto `\n`. Você também pode assumir que o `dictionary` texto conterá pelo menos uma palavra, que nenhuma palavra terá mais de `LENGTH` (uma constante definida em `dictionary.h`) caracteres, que nenhuma palavra aparecerá mais de uma vez, que cada palavra conterá apenas letras minúsculas do alfabeto e possivelmente apóstrofos, e que nenhuma palavra começará com um apóstrofo.
- Pode-se assumir que `check` serão transmitidas apenas palavras que contenham caracteres alfabéticos (maiúsculos ou minúsculos) e, possivelmente, apóstrofos.
- Seu corretor ortográfico só pode aceitar ` `text` and` e, opcionalmente, `as` `dictionary` como entrada. Embora você possa estar inclinado (principalmente se estiver entre os mais familiarizados com o assunto) a "pré-processar" nosso dicionário padrão para derivar uma "função hash ideal" para ele, você não pode salvar o resultado de tal pré-

processamento em disco para carregá-lo de volta na memória em execuções subsequentes do seu corretor ortográfico a fim de obter alguma vantagem.

- Seu corretor ortográfico não deve vazar memória. Certifique-se de verificar se há vazamentos com `valgrind`.
- **A função hash que você escrever deve ser, em última análise, de sua autoria, e não uma que você procure online.**

Certo, prontos para começar?

- Implementar `load`.
- Implementar `hash`.
- Implementar `size`.
- Implementar `check`.
- Implementar `unload`.

Dicas

Clique nos botões abaixo para ler algumas dicas!

▼ **Implement`load`**

Complete a `load` função. `load` Ela deve carregar o dicionário na memória (em particular, em uma tabela hash!). `load` Deve retornar `true` se a operação for bem-sucedida e, `false` caso contrário, retornar falso.

Considere que este problema é composto apenas de problemas menores:

1. Abra o arquivo de dicionário
2. Leia cada palavra no arquivo.
    1. Adicione cada palavra à tabela hash.
3. Feche o arquivo de dicionário

Escreva um pseudocódigo para se lembrar de fazer exatamente isso:

```
bool load(const char *dictionary)
{
    // Open the dictionary file

    // Read each word in the file

        // Add each word to the hash table

    // Close the dictionary file
}
```

Considere primeiro como abrir o arquivo de dicionário. O `fopen` (https://manual.cs50.io/3/fopen)modo `read` é uma escolha natural. Você pode usar o modo `read` `r`, visto que precisa apenas *ler* palavras do arquivo de dicionário (e não *escrevê-las* ou *adicioná* -las).

```
bool load(const char *dictionary)
{
    // Open the dictionary file
    FILE *source = fopen(dictionary, "r");

    // Read each word in the file

        // Add each word to the hash table

    // Close the dictionary file
}
```

Antes de prosseguir, você deve escrever um código para verificar se o arquivo foi aberto corretamente. Isso fica a seu critério! Também é recomendável fechar todos os arquivos que você abrir, então agora é um bom momento para escrever o código para fechar o arquivo do dicionário:

```
bool load(const char *dictionary)
{
    // Open the dictionary file
    FILE *source = fopen(dictionary, "r");

    // Read each word in the file

        // Add each word to the hash table

    // Close the dictionary file
    fclose(source);
}
```

O que resta é ler cada palavra do arquivo e adicioná-la à tabela hash. Retorne `true` quando toda a operação for bem-sucedida e, `false` em caso de falha, retorne o resultado. Considere seguir a resolução passo a passo deste problema e continue a dividir os subproblemas em problemas ainda menores. Por exemplo, adicionar cada palavra à tabela hash pode ser apenas uma questão de implementar algumas etapas ainda menores:

1. Crie espaço para um novo nó de tabela hash.
2. Copie a palavra para o novo nó.
3. Use o hash da palavra para obter seu valor de hash.
4. Insira o novo nó na tabela hash (usando o índice especificado pelo seu valor hash).

É claro que existem outras maneiras de abordar esse problema, cada uma com suas próprias vantagens e desvantagens de projeto. Por essa razão, o restante do código fica a seu critério!

▼ **Implement**`hash`

Complete a `hash` função. `hash` Ela deve receber uma string, `word`, como entrada e retornar um valor positivo (“unsigned”) `int`.

A função hash fornecida retorna um valor `int` entre 0 e 25, inclusive, com base no primeiro caractere de uma palavra `word`. No entanto, existem muitas maneiras de implementar uma função hash além de usar o primeiro caractere (ou *caracteres*) de uma palavra. Considere uma função hash que usa a soma de valores ASCII ou o comprimento de uma palavra. Uma boa função hash reduz as colisões e tem uma distribuição (na maioria das vezes!) uniforme entre os "buckets" da tabela hash.

▼ **Implement`size`**

Complete a `size` função. `size` Ela deve retornar o número de palavras carregadas no dicionário. Considere duas abordagens para este problema:

- Conte cada palavra à medida que a carrega no dicionário. Retorne essa contagem quando `size` a função for chamada.
- A cada `size` chamada, percorra as palavras na tabela hash para contá-las. Retorne essa contagem.

Qual lhe parece mais eficiente? Seja qual for a sua escolha, deixaremos o código a seu critério.

▼ **Implement`check`**

Complete a `check` função. `check` Ela deve retornar verdadeiro `true` se a palavra estiver presente no dicionário, caso contrário , deve retornar falso `false`.

Considere que esse problema também é composto por problemas menores. Se você implementou uma tabela hash, encontrar uma palavra requer apenas alguns passos:

1. Use o hash da palavra para obter seu valor de hash.
2. Pesquise na tabela hash a localização especificada pelo valor hash da palavra.
    1. Retorne `true` se a palavra for encontrada.
3. Retorna `false` se nenhuma palavra for encontrada.

Para comparar duas strings sem distinção entre maiúsculas e minúsculas, você pode achar útil a função `hash` `strcasecmp` (https://man.cs50.io/3/strcasecmp)(declarada em `hash` `strings.h`)! Você provavelmente também vai querer garantir que sua função hash não seja sensível a maiúsculas e minúsculas, de forma que `a` `foo` e `b` `FOO` tenham o mesmo valor de hash.

▼ **Implement`unload`**

Complete a `unload` função. Certifique-se de `free` usar `unload` toda a memória que você alocou `load`!

Lembre-se de que este `valgrind` é o seu novo melhor amigo. Saiba que ele `valgrind` monitora vazamentos de memória enquanto seu programa está em execução, portanto, certifique-se de

fornecer argumentos de linha de comando se quiser `valgrind` analisar o código `speller` enquanto usa um arquivo `dictionary` de texto específico, como no exemplo abaixo. É melhor usar um arquivo de texto pequeno, caso contrário, `valgrind` a execução pode demorar bastante.

```
valgrind ./speller texts/cat.txt
```

Se você executar o comando `valgrind` sem especificar um parâmetro `text` para `for` `speller`, suas implementações de `load` `and` `unload` não serão realmente chamadas (e, portanto, não serão analisadas).

Se tiver dúvidas sobre como interpretar o resultado de `valgrind`, não hesite em pedir `help50` ajuda:

```
help50 valgrind ./speller texts/cat.txt
```

Guias passo a passo

Observe que esta lista de reprodução contém 6 vídeos.

Como testar

Como verificar se o seu programa está identificando corretamente as palavras com erros ortográficos? Bem, você pode consultar as "chaves de resposta" que estão dentro do `keys` diretório que, por sua vez, está dentro do seu `speller` diretório. Por exemplo, dentro de `

<diretório_de_resposta `keys/lalaland.txt`>` estão todas as palavras que o seu programa *deve* considerar como erros ortográficos.

Você poderia, portanto, executar seu programa em algum texto em uma janela, como no exemplo abaixo.

```
./speller texts/lalaland.txt
```

E você poderia então executar a solução da equipe no mesmo texto em outra janela, como no exemplo abaixo.

```
./speller50 texts/lalaland.txt
```

E você poderia então comparar as janelas visualmente, lado a lado. Isso pode se tornar tedioso rapidamente, no entanto. Portanto, talvez seja melhor "redirecionar" a saída do seu programa para um arquivo, como no exemplo abaixo.

```
./speller texts/lalaland.txt > student.txt
./speller50 texts/lalaland.txt > staff.txt
```

Em seguida, você pode comparar os dois arquivos lado a lado na mesma janela com um programa como o `diff`, como mostrado abaixo.

```
diff -y student.txt staff.txt
```

Alternativamente, para economizar tempo, você pode simplesmente comparar a saída do seu programa (presumindo que você o redirecionou para, por exemplo, `student.txt`) com uma das respostas sem executar a solução da equipe, como no exemplo abaixo.

```
diff -y student.txt keys/lalaland.txt
```

Se a saída do seu programa corresponder à da equipe, `diff` serão exibidas duas colunas que devem ser idênticas, exceto, talvez, pelos tempos de execução na parte inferior. Se as colunas forem diferentes, você verá um "\" `>` ou um `|` "\" onde elas diferem. Por exemplo, se você vir

```
MISSPELLED WORDS                                              MISSPELLED WORDS

TECHNO                                                        TECHNO
L                                                             L
                                                            > Thelonious
Prius                                                         Prius
                                                            > MIA
L                                                             L
```

Isso significa que seu programa (cuja saída está à esquerda) não considera que " `Thelonious` or" `MIA` esteja escrito incorretamente, embora a saída da equipe (à direita) considere, como é

sugerido pela ausência de, digamos, `Thelonious` na coluna da esquerda e a presença de `Thelonious` na coluna da direita.

Por fim, certifique-se de testar com os dicionários padrão, tanto o grande quanto o pequeno. Tenha cuidado para não presumir que, se sua solução funcionar corretamente com o dicionário grande, ela também funcionará corretamente com o pequeno. Veja como testar com o dicionário pequeno:

```
./speller dictionaries/small texts/cat.txt
```

Correção

```
check50 cs50/problems/2025/x/speller
```

Estilo

```
style50 dictionary.c
```

Solução da Equipe

Como avaliar a velocidade (e a correção) do seu código? Bem, como sempre, sinta-se à vontade para testar a solução da equipe, como a abaixo, e comparar os resultados com os seus.

```
./speller50 texts/lalaland.txt
```

Como enviar

```
submit50 cs50/problems/2025/x/speller
```