Esse é o **CC50.**
O Curso de Harvard, no Brasil 2012/01

Pset 6: Erros Ortográficos

a ser entregue até: 19:00, sex 20/04

Tenha certeza de que seu código é bem comentado
de forma que a funcionalidade seja aparente apenas pela leitura dos comentários.

Objetivos.

- Leva você a projetar e implementar a sua própria estrutura de dados.
- Otimizar ao máximo o tempo de exceução do seu código (mundo real).
- Desafiar a BIG BOARD!

Leitura recomendada.

- Seções 18 - 20, 27 - 30, 33, 36 e 37 de http://informatica.hsw.uol.com.br/programacao-em-c.htm

Honestidade Acadêmica.

Todo o trabalho feito no sentido do cumprimento das expectativas deste curso deve ser exclusivamente seu, a não ser que a colaboração seja expressamente permitida por escrito pelo instrutor do curso. A colaboração na realização de Psets não é permitida, salvo indicação contrária definida na especificação do Set.

Ver ou copiar o trabalho de outro indivíduo do curso ou retirar material de um livro, site ou outra fonte, mesmo em parte e apresentá-lo como seu próprio constitui desonestidade acadêmica, assim como mostrar ou dar a sua obra, mesmo em parte, a um outro estudante. Da mesma forma é desonestidade acadêmica apresentação dupla: você não poderá submeter o mesmo trabalho ou similar a este curso que você enviou ou vai enviar para outro. Nem poderá fornecer ou tornar as soluções disponíveis para os Psets para os indivíduos que fazem ou poderão fazer este curso no futuro.

Você está convidado a discutir o material do curso com os outros, a fim de melhor compreendê-lo. Você pode até discutir sobre os Psets com os colegas, mas você não pode compartilhar o código. Em outras palavras, você poderá se comunicar com os colegas em Português, mas você não pode comunicar-se em, digamos, C. Em caso de dúvida quanto à adequação de algumas discussões, entre em contato com o instrutor.

Você pode e deve recorrer à Web para obter referências na busca de soluções para os Psets, mas não por soluções definitivas para os problemas. No entanto, deve-se citar (como comentários) a origem de qualquer código ou técnica que você descubra fora do curso.

Todas as formas de desonestidade acadêmica são tratadas com rigor.

Licença.

Notas:

Seu trabalho neste Pset será avaliado em três quesitos principais:

Exatidão. Até que ponto o seu código é consistente com as nossas especificações e livre de bugs?
Design. Até que ponto o seu código é bem escrito*(*escrito claramente, funcionando de forma eficiente, elegante, e / ou lógica)?
Estilo. Até que ponto o seu código é legível (comentado e indentado, com nomes de variáveis apropriadas)?

Começando.

☐ Oi! Abra o Terminal e, como de costume, crie um diretório dentro de cc50 chamado pset6. Extraia todos os arquivos de pset6.zip dentro desse diretório. Agora liste os arquivos dentro de pset6/, você deverá ver o seguinte:

```
Makefile dictionary.c dictionary.h questions.txt speller.c texts
```

O seu trabalho nesse Pset será feito, basicamente, nos arquivos dictionary.c e speller.c. Os arquivos dentro da pasta texts são textos que podem conter problemas ortográficos, a sua missão é fazer um programa que itera através desses textos e descobre todas as palavras que tem, possivelmente, erros de ortografia.

☐ Mas, antes disso, vamos adicionar algo à sua caixa de ferramentas.

`nano` é um dos mais simples editores de texto que podem ser encontrados em um sistema Linux. Um resultado disso, porém, é sua falta de recursos. Assim é hora de passar para algo com uma interface mais interessante. Você sempre soube que pode acessar todos os arquivos e pastas com o Terminal (através de cd) mas também pode acessar os mesmos arquivos e pastas através do gerenciador de arquivos do Ubuntu. Você pode então simplesmente começar a abrir o seu código através da interface gráfica (lê-se contrário de tela preta) com um editor de texto que já vem com o Ubuntu (normalmente o Gedit).

Para esse Pset, você é encorajado a escrever o seu código usando qualquer editor de texto que você gostaria (incluindo `nano` se você se apaixonou) ou outro que pode ser baixado no Centro de Softwares do Ubuntu ou até mesmo um ambiente de desenvolvimento integrado (IDE).

Para ser claro, embora agora você possa escrever o seu código com o Gedit ou outra coisa, você ainda vai querer abrir o Terminal para executar `gcc`, `gdb`, make, `valgrind` e quaisquer outros comandos.

Xei do erro.

☐ Teoricamente, para um input de tamanho *n,* um algoritmo com um tempo de execução *n* é assintoticamente equivalente, em termos de *O, a* um algoritmo com um tempo de execução de 2*n.* No mundo real, porém, o fato é que o último é duas vezes mais lento que o anterior.

O desafio de vocês é implementar o corretor ortográfico mais rápido que você conseguir! Com "mais rápido", porém, estamos falando de segundos, do mundo real, que podem ser percebidos de verdade – nada daquelas coisas assintóticas.

Em `speller.c`, nós escrevemos um programa que foi projetado para verificar a ortografia de um arquivo depois de carregar um dicionário de 143.091 palavras do disco para a memória. Embora tanto os arquivos quanto o dicionário sejam em inglês, você não precisa se preocupar com entender o que há dentro deles, mas simplesmente checar se cada palavra de um arquivo

está presente nesse dicionário. Infelizmente, nós não chegamos a fazer a parte de carregar as palavras. Ou a de verificar o texto. Essas duas coisas (e algumas mais) deixamos para você!

Antes de entender como `speller.c` funciona, vá em frente e abra `dictionary.h` com o Gedit ou qualquer outra coisa. Quatro funções são declaradas nesse arquivo; tome nota do que cada uma deve fazer. Agora abra `dictionary.c`. Repare que nós implementamos as quatro funções, mas bem mal, apenas o suficiente para que o código compile. Seu trabalho nesse Set de Problemas é reimplementar cada uma dessas funções da forma mais inteligente possível, para que esse corretor ortográfico funcione como o enunciado. E funcione rápido!

Vamos começar.

☐ Abra `speller.c` e passe algum tempo olhando o código e os comentários nele. Você não precisa mudar nada nesse arquivo, mas você deve entendê-lo mesmo assim. Observe como, utilizando `getrusage`, estaremos capturando o tempo de execução das suas implementações de `check`, `load`, `size` e `unload`. Note também como nós iteramos através do conteúdo de algum arquivo passando palavra por palavra para `check`. No final, relatamos cada erro de ortografia no arquivo junto com um monte de estatísticas.
Note, aliás, que nós definimos o uso de `speller` da seguinte forma

```
./speller [dict] file
```

onde `dict` deve ser um arquivo contendo uma lista de palavras em letras minúsculas, uma por linha, e `file` é um arquivo a ser corrigido ortográficamente. Como sugerem os colchetes, o uso de `dict` é opcional, se esse argumento for omitido, `speller` usará `/pset6/dict/words` como padrão para seu dicionário. Dentro desse arquivo estão aquelas 143.091 palavras que você deve carregar na memória. Na verdade, dê uma olhada nesse arquivo para ter uma noção de sua estrutura e tamanho. Observe que cada palavra aparece em letras minúsculas (mesmo, por simplicidade, nomes próprios e siglas). De cima para baixo, o arquivo é organizado em ordem alfabética, com apenas uma palavra por linha (cada uma das quais termina com `\n`). Nenhuma palavra é maior do que 45 caracteres, e nenhuma palavra aparece mais de uma vez. Durante o desenvolvimento, você pode achar útil fornecer a `speller` um `dict` feito por você que contém muito menos palavras, para que você não lute para debugar um programa que carrega uma estrutura enorme na memória.

Não passe desse ponto até que você tenha certeza que entendeu como `speller` funciona!

☐ É bem provável que você não tenha passado tempo suficiente olhando `speller.c`. Volte ao tópico anterior e entenda como ele funciona!

☐ Ok, tecnicamente esse último tópico induziu um loop infinito. Mas vamos supor que você saiu fora dele. Em `questions.txt`, responda a cada uma das seguintes perguntas em uma ou mais frases.

0. O que é pneumoultramicroscopicossilicovulcanoconiose?
1. De acordo com a sua página do manual, o que `getrusage` fazer?

2. De acordo com a mesma página, quantos membros tem uma variável do tipo `struct rusage`?
3. Porque você acha que passamos `before` e `after` por referência (em vez de por valor) para `calculate`, mesmo que nós não estejamos mudando os seus conteúdos?
4. Explicar com a maior precisão possível, em um parágrafo ou mais, como `main` age para ler as palavras de um arquivo. Em outras palavras, nos convença de que você realmente entende como o for loop dessa função funciona.
5. Porque você acha que nós usamos `fgetc` para ler cada caracter de cada palavra ao vez ao invés de `fscanf` com uma string formatada como `"%s"` para ler palavras inteiras de uma vez? Dito de outra forma, que problemas poderiam surgir ao depender de `fscanf` para realizar essa tarefa?

☐ Agora dê uma olhada na `Makefile`. Repare que nós empregamos alguns truques novos. Em vez de escrever alvos específicos hard-coded, nós definimos variáveis (não no sentido de C, mas no sentido de uma `Makefile`. A linha abaixo define uma variável chamada `CC` que especifica que make deve usar o `gcc` para compilar.

```
CC = gcc
```

A linha abaixo define uma variável chamada CFLAGS que especifica, por sua vez, que o gcc deve usar algumas flags familiares.

```
CFLAGS = -ggdb -std=c99 –Wall –Werror
```

A linha abaixo define uma variável chamada `EXE`, cujo valor será o nome do nosso programa.

```
EXE = speller
```

A linha abaixo define uma variável chamada `HDRS`, cujo valor é uma lista separada por espaço de arquivos de cabeçalho usados por `speller`.

```
HDRS = dictionary.h
```

A linha abaixo define uma variável chamada `SRCS`, cujo valor é uma lista separada por espaço de arquivos C que irão, juntos, implementar `speller`.

```
SRCS = speller.c dictionary.c
```

A linha abaixo define uma variável chamada `OBJS`, cujo valor é idêntico ao de `SRCS`, exceto que a extensão de cada arquivo não `é .c` mas .o.

```
OBJS = $(SRCS:.c=.o)
```

As linhas abaixo definem um alvo padrão usando essas variáveis.

```
$(EXE): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $(OBJS)
```

A linha abaixo especifica que todos os nossos arquivos `.o` "dependem" de `dictionary.h` de modo que qualquer alteração nele induz a recompilação do código quando você executar `make`.

```
$(OBJS): $(HDRS)
```

Finalmente, as linhas abaixo definem os alvos para limpar o diretório desse Pset.

```
clean:
    rm -f core $(EXE) *.o
```

Saiba que você pode modificar essa `Makefile` como achar melhor. Na verdade, você precisa modificá-la se você criar qualquer arquivo `.c` ou `.h`.
Mesmo que essa `Makefile` seja um pouco mais complicada que do que as anteriores, compilar o seu código continua sendo tão fácil quanto executar

```
make
```

Mas agora você sabe como fazer uma `Makefile` mais sofisticada!

☐ Vamos olhar agora aqueles arquivos de texto! Repare que em `/pset6/texts/` se encontram vários textos com os quais você será capaz de testar o seu `speller`. Entre os arquivos estão o script de *Austin Powers: Um Agente Nada Discreto*, um trecho de um filme do *Monty Python*, três milhões de bytes de Tolstoi, um ótimo trecho do *Guia do Mochileiro das Galáxias,* alguns trechos de Maquiavel e Shakespeare, a Bíblia inteira do Rei James V, e muito mais. Para que você saiba o que esperar, abra e dê uma olhada em cada um desses arquivos.

Agora, como você deve após ter lido `speller.c` cuidadosamente, o output de `speller`, se executado em, digamos, `austinpowers.txt`, deve ser semelhante ao abaixo. Nós omitímos alguns dos nossos favoritos "erros ortográficos". E, para não estragar a diversão, nós temos omitido nossas próprias estatísticas para agora.

```
MISSPELLED WORDS

[...]
Bigglesworth
[...]
Fembots
[...]
Virtucon
[...]
friggin'
[...]
shagged
[...]
trippy
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN FILE:
```

```
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` gasta executando a sua implementação de load. `TIME IN check` representa o número de segundos que `speller` gasta, no total, executando a sua implementação de check. `TIME IN size` representa o número de segundos que `speller` gasta executando a sua implementação de size. `TIME IN unload` representa o número de segundos que `speller` gasta executando a sua implementação de `unload`. `TIME IN TOTAL` é a soma dessas quatro medições.

Aliás, para ficar claro, com "erro ortográfico" queremos dizer que alguma palavra não está no `dicionário` fornecido. "Fembots", por exemplo, poderia muito bem estar em algum dicionário.

☐ Beleza, o seu desafio é implementar `load`, `check`, `size` e `unload` da forma mais eficiente possível, de forma que todos os `tempos` são minimizados. Não é óbvio o que significa minimizar esses tempos, pois essas medidas com certeza variam conforme você muda os valores de `dict` e file. Mas é aí que está o desafio, ou a diversão, desse Set de Problemas. Essa é a sua chance de realmente focar em design e ver resultados. Embora nós encorajemos você a minimizar o espaço, o seu pior inimigo é o tempo. Mas antes de mergulhar no problema, permita-nos dar-lhe algumas especificações.

i. Você não pode alterar `speller.c`.
Ii. Você pode alterar `dictionary.c` (e, de fato, deve a fim de reimplementar `load`, `check`, `size` e `unload`), mas você não pode alterar as declarações das funções `load`, `check`, `size` e `unload`.
Iii. Você pode alterar `dictionary.h`, mas você não pode alterar as declarações das funções `load`, `check`, `size` e `unload`.
Iv. Você pode alterar a `Makefile`.
V. Você pode adicionar funções a `dictionary.c` ou a arquivos de sua própria criação, desde que todo o seu código compile via `make`.
Vi. A implementação de `check` deve ser case-insensitive. Em outras palavras, se `foo` está em `dict`, `check` deve retornar `true` para qualquer possibilidade de capitalização; nenhum desses: `foo`, `foO`, `fOo`, `fOO`, `fOO`, `Foo`, `FoO`, `FOo` e `FOO` deve ser considerado incorreto.
Vii. A sua implementação de `check` só deve retornar `true` quando uma palavra realmente se encontra em `dict`. Você não pode colocar palavras comuns hard-coded no seu código. Além disso, as palavras devem ser exatamente como as encontradas em `dict` para que `check` retorne `true`. Em outras palavras, mesmo que `foo` esteja em `dict`, `check` deve retornar `false` para um argumento `foo's` se `foo's` não está também em `dict`.
Viii. Você pode assumir que a `check` só serão passadas strings com caracteres alfabéticos e/ou apóstrofos.
Ix. Você pode assumir que qualquer `dict` passado para o seu programa será estruturado exatamente como o nosso, ordenado alfabeticamente de cima para baixo com uma palavra por linha, cada uma das quais terminando com `\n`. Você também pode assumir que nenhuma palavra será mais longo do que `LENGTH` (uma constante definida em

`dictionary.h)` caracteres, que nenhuma palavra vai aparecer mais de uma vez, e que cada palavra irá conter apenas caracteres alfabéticos minúsculos e possivelmente apóstrofos.

X. O seu corretor ortográfico só pode aceitar `file` e, opcionalmente, `dict` como input.

xi. Você pode pesquiser sobre funções hash em livros ou na internet, desde que você cite a origem de qualquer função hash que você integrou no seu próprio código.

Tudo bem, pronto para começar?

☐ Implemente `load!`

Permita-nos sugerir que você faça alguns dicionários menores que o padrão de 143.091 palavras para testar seu código durante o desenvolvimento.

☐ Implemente check`!`

Permita-nos sugerir que você tente verificar a ortografia de alguns arquivos pequenos antes de tentar, hmm, na Bíblia por exemplo?!

☐ Implemente `size!`

Se você já tinha pensado nisso ao implementar as outras, essa é fácil!

☐ Implementar `unload!`

Certifique-se de liberar qualquer memória que você alocou em `load!`

☐ Na verdade, tenha certeza que o seu corretor ortográfico não deixa qualquer memory leak. Lembre-se que `valgrind` é o seu mais novo melhor amigo. Saiba que o `valgrind` checa se existe algum memory leak enquanto o seu programa está rodando, por isso não deixe de fornecer os argumentos de linha de comando se você quiser que `valgrind` analise `speller`, como pelo comando abaixo

```
valgrind -v --leak-check=full ./speller texts/austinpowers.txt
```

Se você executar o `valgrind` sem especificar um `file` para `speller`, as suas implementações de `load` e `unload` não vão ser chamadas de fato (e, portanto, analisadas).

☐ Não se esqueça do seu outro bom amigo, `gdb`.

☐ E ajuda@cc50.com.br

☐ Quando o seu programa estiver realmente funcionando e o mais rápido possível, você poderá, se quiser, desafiar a Big Board. Todos os alunos que quiserem podem comparar o seu corretor ortográfico com os dos outros na aula, rodando no mesmo computador, com o mesmo input. Nós faremos então uma competição para ver quem conseguiu implementar o corretor ortográfico mais rápido do CC50!

Nós honraremos aqueles no topo da Big Board.

Na verdade, você pode querer tirar a flag `-ggdb` quando for desafiar a BIG BOARD. E você pode querer ler sobre as flags -O de otimização do gcc.

Aqueles mais confortáveis também pode achar ferramentas como `gprof` e `gcov` interessantes.

☐ Parabéns! Neste ponto, o verificador de ortografia está presumivelmente completo (e rápido!). Então está na hora de algumas perguntas. Em `questions.txt`, responda cada uma das seguintes perguntas em um parágrafo curto.

6. Por que você acha que nós declaramos tantos parâmetros `const` em `dictionary.c` e `dictionary.h`?
7. Qual estrutura de dado(s) você usou para implementar o seu corretor ortográfico? Certifique-se de não responder apenas "hash table", "trie", ou coisa parecida. Fale um pouco sobre o que está dentro de cada um dos seus "nós".
8. Quão lento era o seu código na primeira vez que você o fez funcionar?
9. Que tipos de alterações, se for o caso, você fez no seu código ao longo da semana (er, quinta-feira), a fim de melhorar o seu desempenho?
10. Você sente que o seu código tem alguma parte pesada que você não foi capaz de desbastar?

☐ Esse foi o Pset 6.