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

Pset 3: O Jogo dos Quinze

a ser entregue até: 19:00, sex 16/03

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

Objetivos.

- Introduzir você a programas maiores, com múltiplos arquivos.
- Capacitá-lo a criar Makefiles.
- Apresentá-lo a literatura da ciência da computação.
- Implementar um simpático divertimento.

Leitura recomendada.

- Seção 17 de `http://informatica.hsw.uol.com.br/programacao-em-c.htm`

diff pset3.pdf hacker3.pdf.

- A edição Hacker desafia você a implementar `sort` em $O(n)$ em vez de $O(n^2)$.
- A edição Hacker pede-lhe para brincar de Deus.

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

☐ Abra o seu Terminal e, como de costume, crie um diretório dentro de cc50 chamado hacker3. Extraia todos os arquivos de hacker3.zip dentro desse diretório .

Se você listar o conteúdo do seu diretório de trabalho atual (lembra-se como?), você deve ver algo como o seguinte.

```
fifteen/  find/  hacker3.pdf
```

Como você pode perceber, o seu trabalho para esse Pset será organizado dentro de dois subdiretórios.

☐ Não se esqueça da Lista de Discussões do CC50! Você pode postar perguntas suas ou procurar respostas para as perguntas já feitas por outros. E nunca tenha medo de que as suas perguntas sejam "idiotas".

Find.

☐ Agora vamos mergulhar no primeiro desses subdiretórios. Execute o comando abaixo.

```
cd ~/cc50/hacker3/find/
```

Listando o conteúdo desse diretório, você deve ver o seguinte

```
helpers.c  helpers.h  Makefile  find.c  generate.c
```

Uau, um bocado de arquivos, né? Não se preocupe, nós vamos explicar todos.

☐ Em `generate.c` está implementado um programa que usa `rand` para gerar um monte de números pseudoaleatórios, um por linha. (Lembra-se de `rand` do Pset 1?) Vá em frente e compile este programa, executando o comando abaixo.

```
make generate
```

Agora, execute o programa que você acabou de compilar executando o comando abaixo.

```
./generate
```

Você deve ser informado sobre a utilização adequada do programa:

```
Utilização: generate n [s]
```

Como este resultado sugere, este programa espera um ou dois argumentos de linha de comando. O primeiro, `n`, é obrigatório; ele indica quantos números pseudoaleatórios você gostaria de gerar. O segundo argumento, `s`, é opcional, como os colchetes implicam; se fornecido, ele representa o valor que o gerador pseudoaleatório deve usar como seed. (Lembra-se o que uma

seed é do Pset 1?) Vá em frente e execute este programa novamente, desta vez com um valor de n de, digamos, `10`, como a seguir, você deverá ver uma lista de 10 números pseudoaleatórios.

```
./generate 10
```

Execute o programa pela terceira vez usando esse mesmo valor para `n`, você deverá ver uma lista de 10 números diferentes dos anteriores. Agora tente executar o programa com um valor para `s` (por exemplo `0`), como segue.

```
./generate 10 0
```

Agora execute esse mesmo comando novamente:

```
./generate 10 0
```

Aposto que você viu a mesma sequência de dez números pseudoaleatórios de novo? É isso que acontece se você não variar a semente inicial de um gerador de números pseudoaleatórios.

☐ Agora dê uma olhada no `generate.c` com o Nano. (Lembra-se como?) Os comentários na parte superior explicam a funcionalidade geral do programa. Mas parece que nós nos esquecemos de comentar o código em si. Leia o código com cuidado até que você entenda cada linha e, em seguida, comente o nosso código, substituindo cada `TODO` com uma frase que descreve a finalidade ou funcionalidade da(s) linha(s) correspondente(s) de código. Perceba que um comentário feito com `/*` e `*/` pode se estender por várias linhas enquanto um comentário precedido por `//` só pode se estender até o fim de uma linha, este último é um recurso do C99 (a versão de C que estamos usando) . Se achar que o Pset1 foi há muito tempo atrás, você pode querer ler sobre `rand` e `srand` novamente nas URLs abaixo.

```
http://www.cs50.net/resources/cppreference.com/stdother/rand.html
http://www.cs50.net/resources/cppreference.com/stdother/srand.html
```

Ou você pode executar os comandos abaixo.

```
man rand
man srand
```

Uma vez que você terminou de comentar `generate.c`, `recompile` o programa para ter certeza que você não quebrou nada reexecutando o comando abaixo.

```
make generate
```

Se `gerar` não compilar corretamente, dê uma olhada no que você acabou de fazer e conserte o que você quebrou!

Agora, lembre-se que `make` automatiza a compilação do seu código para que você não tenha que sempre executar `gcc` manualmente junto com um monte de switches. Observe, na verdade, como `make` acaba de executar um comando muito longo para você. No entanto, a medida que seus programas crescem em tamanho, `make` não será mais capaz de descobrir a partir do contexto

como compilar seu código, você vai precisar começar a contar para `make` como seu programa deve ser compilado, principalmente quando ele envolve múltiplas fontes *(*arquivos .c). E por isso vamos começar a confiar em "Makefiles", arquivos de configuração que dizem a `make` exatamente o que fazer.

Como `make` soube como compilar `generate` neste caso? Ele usou um arquivo de configuração que nós escrevemos. Usando Nano, vá em frente e olhe o arquivo chamado `Makefile` que está no mesmo diretório que `generate.c`. Este `Makefile` é, essencialmente, uma lista de regras que nós escrevemos para você que diz `make` como construir `generate a` partir de `generate.c` para você. As linhas relevantes constam abaixo.

```
generate: generate.c
    gcc -ggdb -std=c99 -Wall -Werror -Wformat=0 -o generate generate.c
```

A primeira linha diz a `make` que o "alvo" chamado `generate` deve ser construído, invocando a linha que vem a seguir. Além disso, essa primeira linha diz ao `make` que `generate` é dependente de `generate.c`. Isso faz com que `make` reconstrua `generate` se o arquivo `generate.c` foi modificado desde a última utilização de `make`. Bom truque para economizar o tempo, né? Vá em frente e execute o comando abaixo novamente, supondo que você não tenha modificado `generate.c`.

```
make generate
```

Você deve ser informado de que `generate` já está atualizado. Aliás, saiba que o espaço em branco inicial da segunda linha de `make` não é uma sequência de espaços, mas sim um tab. Infelizmente, `make` requer que os comandos sejam precedido por tabs, por isso tome cuidado para não alterá-los para espaços com o Nano (que converte automaticamente tabs para quatro espaços), se não você pode encontrar erros estranhos! A flag `-Werror`, lembre-se, diz ao `gcc` para tratar avisos (ruins) como se fossem erros (muito pior) a fim de que você seja forçado (em uma boa e instrutiva maneira!) a corrigi-los.

☐ Agora dê uma olhada em `find.c` com o Nano. Note que este programa espera um argumento de linha de comando único: a "agulha" para pesquisar em um "palheiro" de valores. Uma vez que você já tenha olhado todo o código, vá em frente e compile o programa executando o comando abaixo.

```
make find
```

Note que `make`, na verdade, executou o seguinte comando para você:

```
gcc -ggdb -std=c99 -Wall -Werror -Wformat=0 -o find find.c helpers.c -lcc50 -lm
```

Observe, ainda, que você acaba de compilar um programa que inclui não um, mas dois arquivos .c: `helpers.c` e `find.c`. Como é que `make` sabe o que fazer? Bem, mais uma vez, abra a `Makefile` para ver a mágica por trás dos panos. As linhas relevantes aparecem abaixo.

```
find: find.c helpers.c helpers.h
    gcc -ggdb -std=c99 -Wall -Werror -Wformat=0 -o find find.c helpers.c -lcc50 -lm
```

Como você pode ver, na primeira linha (após os dois pontos), qualquer alteração em `find.c`, `helpers.c`, ou `helpers.h` vai obrigar `make` a recompilar `find` na próxima vez que ele for chamado.

Vá em frente e execute `find`.

```
./find 13
```

Você será solicitado a fornecer algum “palheiro” (alguns inteiros), uma "palha" de cada vez. Assim que você se cansar de fornecer números inteiros, pressione Ctrl-D para enviar o programa um caractere EOF (End Of File – Fim de Arquivo). Esse caractere vai obrigar `getInt` da Biblioteca do CC50 a retornar `INT_MAX`, uma constante que, pelo código de `find.c`, vai obrigar find a parar de solicitar palha. O programa irá então procurar a agulha no palheiro que você forneceu, relatando, no fim, se a agulha se encontra nesse palheiro. Em suma, este programa procura um valor em um array.

Acontece que você pode automatizar esse processo de fornecimento de feno fornecendo o output de generate ao input de find. Por exemplo, o comando abaixo passa 1.024 números pseudoaleatórios para find, que então procura 13 dentre esses valores.

```
./generate 1024 | ./find 13
```

Alternativamente, você pode "redirecionar" o output de `generate` para um arquivo com um comando como o abaixo.

```
./generate 1024 > numbers.txt
```

Você pode então redirecionar o conteúdo desse arquivo para o input de `find` com o comando abaixo.

```
./find 13 < numbers.txt
```

Vamos terminar olhando para aquela `Makefile`. Observe a linha abaixo.

```
all: find generate
```

Esta linha significa que você pode construir tanto `generate` quanto `find` simplesmente executando o abaixo.

```
make all
```

Ainda melhor, você pode chegar ao mesmo resultado (porque `make` sem nenhum outro argumento sempre executa a primeira linha de uma Makefile) utilizando simplesmente:

```
make
```

Se você pudesse deletar todos os arquivos desse Pset com um único comando! Finalmente, observe essas últimas linhas da `Makefile`:

```
clean:
    rm -f *.o a.out core find generate
```

Estas linhas permitem que você apague todos os arquivos terminados em `.o` ou chamados `a.out`, `core` (tsk, tsk), `find` ou `generate` simplesmente através do comando abaixo.

```
make clean
```

Tenha cuidado para não adicionar, por exemplo, `*.c` à última linha da `Makefile!` (Porquê?) Aliás, qualquer linha que começa com `#` é apenas um comentário.

☐ E agora começa a diversão! Note que `find.c`, chama `sort` uma função declarada em `helpers.h`. Infelizmente nós nos esquecemos de implementar essa função em `helpers.c!` Dê uma olhada em `helpers.c` com o Nano, e você verá que `sort` retorna imediatamente, mesmo que a função `main de find` passe-a um array real. Nos poderíamos ter colocado o conteúdo de `helpers.h` e `helpers.c` em `find.c`. Mas as vezes organizar programas em vários arquivos é muito melhor, especialmente quando algumas funções (por exemplo `sort`) sejam de grande utilidade, que possam ser úteis mais tarde para outros programas, bem como as funções da Biblioteca do CC50.

A propósito, recorde a sintaxe para declarar um array. Você pode não só especificar o tipo de array, mas também especificar o seu tamanho entre colchetes, tal como fazemos com hay`stack` em `find.c`:

```
int haystack[HAY_MAX];
```

Mas ao passar um array para um função, você só especifica o seu nome, assim como fazemos ao passar haystack para sort em `find.c`:

```
if (!sort(haystack, size))
{
    printf("Não foi possível ordenar esse array.\n");
    return 2;
}
```

(Por que nós também passamos o tamanho desse array separadamente?)

Quando você declarar uma função que recebe um array unidimensional como um argumento, porém, você não precisa especificar o tamanho do array, assim como nós não o fazemos ao declarar `sort` em `helpers.h` (e `helpers.c`):

```
bool sort(int values[], int n);
```

Agora vá em frente e implemente `sort` para que a função realmente organize, do menor para o maior, o array de números passado a ela, de tal forma que o seu tempo de execução seja de *O*(*n*), onde *n* é o tamanho do array.[1] Sim, esse tempo de execução é possível, hax0r, porque você pode assumir que cada número do array será não-negativo e menor do que `LIMIT`, uma constante definida em `generate.c`. No entanto perceba que esse array pode possuir números duplicados. Tome cuidado para não alterar a nossa declaração de sort, que deve permanecer:

```
bool sort(int values[], int n);
```

Como este tipo de retorno `bool` indica, esta função não deve retornar um array ordenado, mas deve ordenar "destrutivamente" o próprio array que é passado, movendo os valores nele, no fim retornando `true` se e somente se o array foi ordenado corretamente. (Provavelmente o array sempre será corretamente ordenado, mas, se você utilizar certos truques, é possível, por exemplo, ficar sem memória suficiente. Como discutiremos na Semana 4, arrays não são passados "por valor", mas "por referência", o que significa que à `sort` não será passada a cópia de um array, mas o verdadeiro array original.

Deixamos para você a determinação de como testar a implementação de sort. Mas não se esqueça que `printf` e `gdb` são seus amigos. E não se esqueça que você pode gerar a mesma sequência de números pseudoaleatórios várias vezes especificando a seed de `generate` explicitamente. Antes de enviar, no entanto, não se esqueça de remover todas essas utilizações de `printf`. Nós gostamos dos outputs dos nossos programas do jeito que eles são!

☐ Precisa de ajuda? Vá direto para cc50.com.br/forum!

☐ Agora que `sort` (supostamente) funciona, é hora de desenvolver `search`, a outra função que vive em `helpers.c`. Repare que a nossa versão implementa uma "busca linear", na qual `search` procura por `value` iterando sobre os inteiros em `array` linearmente, da esquerda para a direita. Mande pro espaço as linhas que nós escrevemos e reimplemente `search` utilizando pesquisa binária, que usa a estratégia "dividir e conquistar" que foi empregada na Semana 0, a fim de pesquisar nomes na lista telefônica.[2] Você está convidado a utilizar uma abordagem iterativa ou recursiva. Se você escolher o último, porém, saiba que você não pode mudar a nossa declaração de search, mas você pode escrever uma nova função recursiva (que talvez usa parâmetros um pouco diferentes) que é chamada por `search`.

[1] Tecnicamente, porque nós limitamos com uma constante a quantidade de palha que `find` aceita (e porque o valor do segundo parâmetro de `sort` está limitado pelos 32 bits de um `int`) o tempo de execução de `sort`, não importa como ele seja implementado, é de fato *O*(1). Mesmo assim, pela beleza desse desafio assintótico, pense no tamanho do input de `sort` como *n*.

[2] Não há necessidade de rasgar qualquer coisa pela metade.

O jogo começa.

☐ Agora é hora de jogar. O Jogo dos Quinze é um quebra-cabeça jogado em um tabuleiro bidimensional quadrado com peças numeradas, que escorregam. O objetivo deste quebra-cabeça é organizar todas as peças do tabuleiro, da menor para a maior, da esquerda para a direita, de cima para baixo, com um espaço vazio no canto inferior direito, como mostrado abaixo.[1]

Deslizar qualquer peça que está na vizinhança do espaço vazio constitui um "movimento". Apesar de a configuração acima mostrar um jogo já ganho, observe como as peças 12 e 15 podem ser deslizadas para o espaço vazio. Peças não podem ser movidas diagonalmente ou removidas a força do tabuleiro. Abaixo se encontra uma configuração a partir da qual o quebra-cabeça é solucionável.

[1] Figura retirada de `http://en.wikipedia.org/wiki/Fifteen_puzzle`.

☐ Navegue para `~/cc50/hacker3/fifteen/` e dê uma olhada em `fifteen.c` com o Nano. Dentro deste arquivo está uma base completa para o Jogo dos Quinze.

Implemente God Mode para este jogo.

Primeiro implemente `init` de tal forma que o tabuleiro seja inicializado com uma configuração pseudoaleatória, mas possível de ser resolvida.[1] Em seguida, termine a implementação de `draw`, `move`, e `won` de forma que um ser humano possa realmente jogar o jogo. Mas incorpore no jogo um cheat, segundo o qual, ao invés de digitar um número inteiro entre 1 e d^2 - 1 onde d é a altura e a largura do tabuleiro, o ser humano também pode digitar

```
GOD
```

para obrigar "o computador" a assumir o controle do jogo e resolvê-lo (usando qualquer estratégia, ideal ou não ideal), fazendo, digamos, quatro movimentos por segundo para que o ser humano possa assistir o jogo sendo resolvido. Presumivelmente, você vai precisar trocar `getInt` por algo mais versátil. Está tudo bem se a sua implementação do God Mode só funciona (suportavelmente rápido) para `d <= 4;` você não precisa se preocupar em testar o seu God Mode para `d >` 4. Ah, e você não pode implementar o God Mode, lembrando como `o init` inicializou o tabuleiro (como por lembrar a sequência de movimentos que o seu programa fez para chegar a algum estado pseudoaleatório, mas solucionável). Isso seria, hum, trapacear. Em trapacear.

Para testar a sua implementação, você certamente pode tentar jogar, com ou sem God Mode ativado. (Saiba que você pode encerrar o seu programa pressionando ctrl-c.) Tenha certeza que você (e nós) não pode travar o seu programa, ao fornecer números falsos de peças.

Todas as decisões de design não prescritas aqui explicitamente *(*por exemplo quanto espaço você deve deixar entre os números para a impressão do tabuleiro) são intencionalmente deixadas para você. O tabuleiro, quando impresso deve se parecer, presumivelmente, com o abaixo (possivelmente com outras configurações), mas deixamos esse design para você implementar!

```
15 14 13 12

11 10  9  8

 7  6  5  4

 3  1  2  _
```

Ainda lembre-se que as posições das peças 1 e 2 só devem começar trocadas se o tabuleiro tem um número ímpar de peças (como no caso do tabuleiro 4 x 4 do exemplo acima). Se o tabuleiro tem um número par de peças, essas posições não devem começar trocadas. Por exemplo como no exemplo 3 x 3 abaixo:

[1] Para ser claro, enquanto a edição padrão deste Pset exige que o tabuleiro seja inicializado sempre com uma configuração padrão específica, esta edição Hacker requer que ele seja inicializado com uma configuração pseudoaleatória, mas ainda solucionável.

```
8  7  6

5  4  3

2  1  _
```

Sinta-se livre para ajustar o argumento apropriado para `usleep` para acelerar ou desacelerar a animação. Na verdade, você é bem-vindo a alterar a estética do jogo. Para se divertir (opcionalmente) com "sequências de escape ANSI", incluindo a cor, dê uma olhada em nossa implementação e confira a URL abaixo para mais truques.

```
http://isthe.com/chongo/tech/comp/ansi_escapes.html
```

Você está livre para escrever suas próprias funções e até mesmo alterar as declarações das funções que nós escrevemos. Mas pedimos que você não altere o fluxo da lógica em `main` para que possamos automatizar alguns testes do seu programa, uma vez submetido. Em particular, `main` só retorna `0` quando o usuário realmente ganhou o jogo; outros valores diferentes de zero devem ser retornados em qualquer caso de erro, como implicado no nosso código já pronto. Em caso de dúvida sobre se alguma decisão de design pode contrariar os nossos desejos, mande um e-mail para ajuda@cc50.com.br.

Falando em God Mode, por onde começar? Bem, primeiro leia sobre o jogo dos Quinze. Wikipedia é provavelmente um bom ponto de partida (somente em inglês) :

```
http://en.wikipedia.org/wiki/N-puzzle
```

Em seguida, mergulhe um pouco mais, talvez lendo sobre um algoritmo chamado A* (A-estrela).

```
http://www.policyalmanac.org/games/aStarTutorial_port.htm
```

Considere o uso da "distância de Manhattan" como heurística para a sua implementação. Se você achar que o A* ocupa muita memória (particularmente para $d \geq 4$) você pode querer dar uma olhada no aprofundamento iterativo de A* (IDA*) em vez disso (complicado):

```
http://webdocs.cs.ualberta.ca/ ~ tony/RecentPapers/pami94.pdf
```

Uma outra implementação, talvez até mais simples, pode ser feita como descrito nesse trabalho:

```
http://www.eng.unt.edu/ian/pubs/saml.pdf
```

Você é bem vindo a expandir a sua busca por ideias além dessas nesse documento, mas tome cuidado para que sua pesquisa não o leve a código real. Olhar um pouco de pseudocódigo dos outros é bom, mas, por favor saia do site se você tropeçar em implementações reais de programas para resolver esse problema (seja em C ou em outras linguagens).

☐ Tudo bem, você consegue, implemente este jogo!