MAC2166 – Introdução à Ciência da Computação

Escola Politécnica – Computação / Elétrica – Primeiro Semestre de 2023

Exercício-Programa 3 (EP3)
Data de Entrega: **2 de julho de 2023**

Para se preparar bem para o desenvolvimento de seu EP3, cuja descrição inicia-se na próxima página, leia com atenção as instruções abaixo.

- Utilize **somente** os recursos da linguagem que aprendeu nas aulas.
- Veja em `https://www.ime.usp.br/~mac2166/infoepsC/` as instruções de entrega dos exercícios-programa e atente para as instruções de preenchimento do cabeçalho do seu programa.
- Caso você tenha dúvidas sobre eventuais erros e *warnings* que o compilador produza ao processar o seu programa, consulte o FAQ sobre compilação em `https://www.ime.usp.br/~mac2166/compilacao/`.
- Sempre compile seus programas com as opções **-Wall -ansi -pedantic -O2**. Como nesse programa você deverá usar a função `log()` da biblioteca `math`, use adicionalmente a opção `-lm` para compilar seu programa. A função `log()` recebe um valor double e devolve o logaritmo na base natural deste valor.
- Seu programa deve:
  - funcionar para qualquer entrada que está de acordo com o enunciado (não é necessário verificar se a entrada está "bem formada");
  - estar em conformidade com o enunciado;
  - estar bem estruturado;
  - ser de fácil compreensão, com o uso padronizado da linguagem C.

EP3: Criptografia de imagens

1 Introdução

Neste EP você implementará um programa em C para criptografar e descriptografar imagens. Iremos considerar uma imagem em preto e branco, formada por diversos "pixels" em níveis de cinza, sendo que o tom de cada pixel varia de 0 (preto) a `MaxVal` (branco), que pode ser no máximo 255. Seu EP deverá ser capaz de realizar as tarefas abaixo, ao receber uma imagem da entrada padrão:

- Codificar a imagem utilizando uma senha;
- Decodificar a imagem utilizando uma senha;
- Decodificar a imagem testando todas as senhas possíveis até um dado valor.

Antes de discutirmos como deve ser feita a codificação e a decodificação de uma imagem, vamos descrever o formato de imagem utilizado.

2 Formato PGM

Utilizaremos o formato PGM (*Portable Gray Map*) para armazenar imagens em arquivos. Segundo esse formato, o arquivo deve conter um cabeçalho e a matriz correspondente à imagem. Veja um exemplo a seguir.

```
P2
5 4
16
 9  4  5  0  8
10  3  2  1  7
 9  1  6  3 15
 1 16  9 12  7
```

A primeira linha do arquivo contém a palavra-chave "`P2`", que é obrigatória e está presente em todos os arquivos do tipo PGM que vamos tratar. A segunda linha contém dois inteiros que correspondem às quantidades de colunas e linhas da matriz, respectivamente.[1] A terceira linha contém um inteiro que é o maior valor em um pixel da imagem, `MaxVal`, que pode ser no máximo 255. Os demais números do arquivo correspondem aos tons de cinza da imagem armazenados em forma de uma matriz de inteiros. Cada tom de cinza é um número entre 0 e `MaxVal`, com 0 indicando "preto" e `MaxVal` indicando "branco".

[1] Note que a quantidade de colunas aparece antes da quantidade de linhas.

O formato PGM também permite colocar comentários. Caracteres após o caractere '`#`' até o próximo fim de linha (caractere '`\n`') são comentários e são ignorados. Um exemplo de imagem com comentários:

```
P2
# imagem: exemplo.pgm
5 4
16
009 004 005 000 008
010 003 002 001 007
009 001 006 003 015
001 016 009 012 007
```

Neste EP, por simplicidade, *você pode supor que os arquivos que serão manipulados não contêm comentários*. De fato, vamos fornecer um programa que "filtra" (joga fora) tais comentários. Assim, se você quiser trabalhar com imagens com comentários, você pode primeiro "filtrar" esses comentários e criar um novo arquivo, sem comentários, que pode então ser usado em seu EP.

Observe ainda que o exemplo acima mostra que os inteiros em um arquivo PGM podem ser dados com "zeros à esquerda". Isso não é problema, pois o formato `%d` em `scanf()` ignora tais zeros.

3 Codificação e decodificação

Seja D uma imagem com ℓ linhas e c colunas (um total de $c\ell$ pixels), dada em um arquivo PGM. Vamos armazenar os pixels de D em um vetor de inteiros `image[]`, com os pixels de D em ordem, linha por linha.

Para realizar a codificação e decodificação com uma senha `key`, utilizaremos a função `srand2166(key)`, que inicializa uma função `rand2166()` com a "semente" `key`. Essas funções `srand2166()` e `rand2166()` são como `srand()` e `rand()`, que servem para gerar números "psedoaleatórios". Introduzimos `srand2166()` e `rand2166()` para termos completo controle do que estamos fazendo (evitamos assim de usar `srand()`/`rand()`, que têm comportamentos diferentes em sistemas diferentes). Sempre que o programa for executado, devido ao comando `srand2166(key)`, uma sequência de chamadas à função `rand2166()` irá gerar sempre a mesma sequência de inteiros "pseudoaleatórios". Veja o programa `rand2166.c` disponibilizado no e-Disciplinas, e veja na Seção 6 como deve ser a estrutura geral de seu programa `ep3.c`.

3.1 Codificação

Para codificar uma imagem D, aplicaremos um "filtro" em cada pixel `image[i]`, em ordem, começando de $i = 0$ até $i = c\ell - 1$. Para aplicar o filtro, alteramos o valor do pixel atual adicionando um valor aleatório a ele, mas garantindo que o novo valor seja um inteiro em

$\{0, 1, \ldots,$ `MaxVal`$\}$:

```
(image[i] + (rand2166() % (MaxVal + 1))) % (MaxVal + 1).
```
(1)

Para consolidar o entendimento do filtro acima, vamos aplicá-lo na imagem `exemplo.pgm` descrita na Seção 2, que tem 20 pixels e digamos que esteja armazanada no vetor `image[]`, como abaixo.

```
image[] = |9|4|5|0|8|10|3|2|1|7|9|1|6|3|15|1|16|9|12|7|
```

Vamos codificar `image[]` utilizando a senha `2166`. Assim, vamos em nosso código executar o comando `srand2166(2166)` e depois codificar os pixels `image[i]` para $i = 0, 1, \ldots, 19$. Note que, para esta imagem, `MaxVal` é 16. Na tabela abaixo, temos na segunda coluna os inteiros obtidos na sequência das primeiras 20 chamadas a `rand2166()` executadas após a execução de `srand2166(2166)`; na terceira coluna temos o resto da divisão desses inteiros por 17; na última coluna temos os valores dos pixels após a aplicação do filtro, i.e., temos os valores de todos os pixels da imagem obtida após a codificação.

i	rand2166()	% 17	image[i]	(image[i] + (rand2166() % 17) % 17
0	36403962	9	9	1
1	1956033586	3	4	7
2	1376811626	3	5	8
3	936701757	6	0	6
4	2091297389	8	8	16
5	570366474	1	10	11
6	1929811957	7	3	10
7	904040658	4	2	6
8	764536481	3	1	4
9	1169976166	7	7	14
10	1429150030	14	9	6
11	119962515	9	1	10
12	1870328719	7	6	13
13	1896639094	9	3	12
14	1713480437	15	15	13
15	709998389	2	1	3
16	1523781191	12	16	11
17	1447986662	0	9	9
18	1027140430	5	12	0
19	1675652424	11	7	1

Se armazenarmos os novos valores dos pixels no próprio vetor `image[]`, ficamos com

```
image[] = |1|7|8|6|16|11|10|6|4|14|6|10|13|12|13|3|11|9|0|1|
```

Assim, a imagem codificada é como abaixo:

```
P2
```

```
5 4
16
001 007 008 006 016
011 010 006 004 014
006 010 013 012 013
003 011 009 000 001
```

3.2 Decodificação

Seja D^* a imagem obtida após aplicarmos o processo de codificação descrito na Seção 3.1 acima. Tendo o conhecimento da senha `key` utilizada na codificação, basta aplicar o filtro abaixo em cada pixel `image[i]`, na mesma ordem utilizada na codificação, após executar `srand2166(key)` com a mesma senha `key`:

```
(image[i] - (rand2166() % (MaxVal + 1)) + (MaxVal + 1)) % (MaxVal + 1).    (2)
```

4 Decodificação por força bruta

Para decodificar uma imagem D^* sem que tenhamos a senha, podemos testar todas as senhas possíveis. Porém, um problema claro é como identificar se a imagem decodificada corresponde à imagem original D.

Seja N a quantidade de pixels de uma imagem I. Para $i \in \{0, \ldots, \texttt{MaxVal}\}$, denote por f_i a quantidade de pixels de I iguais a i e defina $p_i = f_i/N$. Para identificar a imagem decodificada, analisaremos a *entropia* $H(I)$ de cada imagem gerada, definida como

$$H(I) = \sum p_i \cdot \ln(1/p_i),$$

em que a soma é sobre todos os índices $i \in \{0, \ldots, \texttt{MaxVal}\}$ tais que $p_i > 0$.

No caso de uma imagem "real" D, o valor de $H(D)$ tende a ser "menor" do que a entropia $H(I)$ de imagens I que foram obtidas através de decodificação com senha incorreta. Assim, a imagem original D terá a entropia notadamente menor do que a entropia das imagens obtidas pelo processo de decodificação com senha incorreta.

5 Organização das funções

Neste EP você deve implementar as funções abaixo, como descrito em suas definições. Você pode, se desejar, implementar outras funções para resolver o EP. Nas funções abaixo, `col` e `row` são, respectivamente, as quantidades de colunas e linhas da imagem armazenada em `image[]`. Ademais, `v_max` corresponde a `MaxVal`, e `key` é a senha com a qual a imagem será codificada/decodificada. Para a codificação e decodificação, serão permitidas como senha os inteiros $0, 1, \ldots, 2^{31} - 1$ (o que acontece com uma imagem codificada com as senhas 0 e $2^{31} - 1$?).

A função `output_image()` abaixo deve imprimir a imagem `image[]` no formato PGM na saída padrão.

```
void output_image(int col, int row, int v_max, int image[]);
```

A função `read_image()` abaixo deve ler uma imagem da entrada padrão, armazenando a quantidade de colunas em `*col`, a quantidade de linhas em `*row`, o valor de `MaxVal` em `*v_max` e a imagem deve ser armazenada no vetor `image[]`, como descrito na Seção 3.

```
void read_image(int *col, int *row, int *v_max, int image[]);
```

A função `encode()` abaixo deve codificar a imagem `image[]` utilizando a senha `key` e armazenar a imagem resultante no próprio vetor `image[]`.

```
void encode(int col, int row, int v_max, int image[], int key);
```

A função `decode()` abaixo deve decodificar a imagem `image[]` utilizando a senha `key` e armazenar a imagem resultante no vetor `decoded[]`.

```
void decode(int col, int row, int v_max, int image[], int key, int decoded[]);
```

A função `decode_in_loco()` abaixo deve decodificar a imagem `image[]` utilizando a senha `key` e armazenar a imagem resultante no próprio vetor `image[]`.

```
void decode_in_loco(int col, int row, int v_max, int image[], int key);
```

Note que a função `decode_in_loco()` pode ser implementada com uma simples chamada à função `decode()`.

A função `decode_brute_force()` abaixo deve decodificar a imagem `image[]` sem ter o conhecimento da senha usada na codificação, tentando todos os valores de senha em $\{0, 1, \ldots,$ `max_key`$\}$, armazenando a imagem resultante no próprio vetor `image[]`.

```
void decode_brute_force(int col, int row, int v_max, int image[], int max_key);
```

A função `entropy()` abaixo deve devolver a entropia da imagem `image[]` com `N` pixels.

```
double entropy(int v_max, int N, int image[]);
```

6 Execução

O seu programa deve receber uma *opção* na *linha de comando*, que pode ser o inteiro 0, 1 ou 2, indicando respectivamente se o programa deve codificar, decodificar com senha, ou decodificar por força bruta. Ademais, o programa deve receber uma senha `key` (inteiro não-negativo), que é a senha utilizada para codificar e decodificar no caso das *opções* 0 e 1, e `key` é a maior senha testada no caso da opção de decodificar por força bruta, *opção* 2.

Compile o seu programa executando `gcc -Wall -ansi -pedantic -O2 -o EP3 ep3.c -lm`, gerando o executável `EP3`. A estrutura geral de seu programa `ep3.c` deve ser como a seguir.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX_PIXELS 1000000
#define rand2166() (rand2166f(-1))
#define srand2166(seed) (rand2166f((seed)))
#define AA 16807
#define MM 2147483647

void output_image(int col, int row, int v_max, int image[]);
void read_image(int *col, int *row, int *v_max, int image[]);
void encode(int col, int row, int v_max, int image[], int key);
void decode(int col, int row, int v_max, int image[], int key, int decoded[]);
void decode_in_loco(int col, int row, int v_max, int image[], int key);
void decode_brute_force(int col, int row, int v_max, int image[], int max_key);
double entropy(int v_max, int N, int image[]);
int rand2166f(int seed);

int main(int argc, char *argv[]) {
  int col, row, v_max;
  int image[MAX_PIXELS];
  int key = atoi(argv[2]);

  /* COLOQUE SEU CÓDIGO AQUI */

  return 0;
}

/* IMPLEMENTAÇÕES DAS FUNCÕES */

int rand2166f(int seed) {
  static long r;
  if (seed >= 0) {
    r = seed;
    return 0;
  }
  r = (r * AA) % MM;
  return r;
}
```

Ao executar seu programa para **codificar** uma imagem `eye.pgm` com a senha `2166`, você deve executar o comando abaixo, em que a imagem codificada será armazenada no arquivo `eyeC.pgm`.

```
./EP3 0 2166 < eye.pgm > eyeC.pgm
```

Ao executar seu programa para **decodificar** uma imagem `eyeC.pgm` que foi codificada com a senha `2166`, você deve executar o comando abaixo, em que a imagem decodificada será armazenada no arquivo `eyeD.pgm`.

```
./EP3 1 2166 < eyeC.pgm > eyeD.pgm
```

Ao executar seu programa para **decodificar** uma imagem `mistery_small.pgm`, codificada com uma senha desconhecida de até 4 dígitos, você deve executar o comando abaixo, que tenta decodificar a imagem com todas as senhas entre 0 e 9999. A imagem decodificada será armazenada no arquivo `revealed.pgm`.

```
./EP3 2 9999 < mystery_small.pgm > revealed.pgm
```

Os arquivos `eye.pgm`, `eyeC.pgm`, `eyeD.pgm`, `mystery_small.pgm` dos exemplos acima estão disponibilizados no e-Disciplinas. A execução do último exemplo acima pode demorar um pouco.

Seu programa deve ser capaz de codificar as imagens `exemplo.pgm`, `eye.pgm`, `escher_hands.pgm`, e também decodificar com senha os arquivos `exemploC.pgm`, `eyeC.pgm`, `escher_handsC.pgm`, codificados com a senha `2166`. Ademais, seu programa deve ser capaz de decodificar sem senha a imagem `mystery_small.pgm`, que foi codificada com uma senha de até 4 dígitos.

7 Pontos extras

Para ganhar 0.5 ponto extra, você deve resolver um "problema" que talvez ocorra em seu programa: para imagens não tão pequenas, como a imagem `escher_stairs.pgm`, talvez seu programa não consiga codificar nem decodificar com senha. Uma pequena alteração em seu programa resolverá esse problema (quantos pixels tem `escher_stairs.pgm`?). Entretanto, apenas com essa alteração simples, seu programa provavelmente não conseguirá decodificar *sem senha* versões codificadas de `escher_stairs.pgm` (isto é, a opção 2 não funcionará). Para ganhar 0.5 ponto extra, seu programa deve ser capaz de

- decodificar, por força bruta (opção 2), versões codificadas da imagem `escher_stairs.pgm` que foram codificadas com uma senha de até 6 dígitos, dentro de um tempo razoável (da ordem de um minuto).

- decodificar, por força bruta (opção 2), a imagem `mystery.pgm`, que foi codificada com uma senha de até 6 dígitos, dentro de um tempo razoável (da ordem de um minuto).

Para ter sucesso na tarefa acima, você precisará implementar a função `decode_brute_force()` usando uma ideia adicional, para que ela seja mais rápida.

Se em vez de ganhar 0.5 ponto extra, você quiser ganhar 1.0 ponto extra, seu programa deve funcionar para imagens ainda maiores. Você deverá alocar os vetores dinamicamente. Seu programa deve ser capaz de

- decodificar, por força bruta (opção 2), `mystery_big.pgm`, que foi codificada com uma senha de até 6 dígitos, dentro de um tempo razoável (da ordem de um minuto).

Observações importantes

Se você for enviar um programa para ganhar 0.5 ou 1.0 ponto extra, leia as observações a seguir.

- Você deve submeter o seu arquivo com o código referente ao ponto extra separadamente do arquivo referente ao seu EP3. Portanto, você deverá submeter **DOIS arquivos**, em entradas diferentes no e-Disciplinas: entregue seu EP3 no local referente ao EP3 `:-)` e entregue seu "EP3 incrementado" no local "EP3 – Ponto Extra".

- Para ter sucesso no código referente ao ponto extra, seu `ep3.c` pode desviar levemente do especificado na página 7. Você pode inclusive mudar o protótipo de uma função, desde que ela execute a tarefa descrita na Seção 5.