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

Trabalho 1

Esta é a especificação do primeiro projeto da disciplina CSF13 - Fundamentos de Programação 1, profs. Bogdan T. Nassu e Leyza B. Dorini, para o período 2023/1.

I) Equipes

O trabalho é individual. A discussão entre colegas para compreender a estrutura do trabalho é recomendada e estimulada, mas cada um 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 17/05/2023, 23:59. Trabalhos entregues após esta data terão sua nota final reduzida em 0,00025% para cada segundo de atraso. Cada estudante deve entregar, através da página da disciplina no Classroom, dois arquivos (separados, sem compressão):

- Um arquivo chamado *t1-x.c*, onde *x* é o seu número de matrícula. O arquivo deve conter as implementações das funções pedidas (com o cabeçalho idêntico ao especificado). Exceto afirmação em contrário, as funções pedidas não devem fazer uso de funções da biblioteca-padrão. Você também não pode usar vetores, matrizes ou outros recursos da linguagem C que não tenham sido explorados em aula antes da especificação deste trabalho. O autor do arquivo deve estar identificado 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 *t1-x.pdf*, onde *x* é o seu número de matrícula. Este arquivo deve conter um relatório breve (em torno de 1 a 2 páginas), descrevendo os desafios encontrados e a forma como eles foram superados. Não é preciso seguir uma formatação específica. O autor do arquivo deve estar identificado 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 uma implementação criada pelos professores, sem “truques” ou otimizações sofisticadas. Os resultados dos trabalhos serão comparados àqueles obtidos pela referência. Os seguintes pontos serão avaliados:

III.a) Compilação. Cada erro que impeça a compilação do arquivo implica em uma penalidade de até 20 pontos. Certifique-se que seu código compila!!!

III.b) Corretude. Sua solução deve 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 será multiplicada por um valor: até 0.7 se o erro for facilmente detectável em testes simples, ou 0.85 do contrário.

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. 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 varías inicializações e incrementos, mas sem bloco de comandos. Evite também replicar trechos 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 15%.

IV) Avaliação (especial)

Indícios de fraude podem levar a uma avaliação especial, com o aluno sendo convocado e questionado 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 final do trabalho será igual à soma das notas de cada função. A nota máxima de cada função é listada junto à sua descrição. A nota máxima para o relatório é 5 pontos.

VI) Apoio

Caso tenha dúvidas, procure a monitoria. Os professores também estarão disponíveis para tirar dúvidas a respeito do projeto, 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) Arquivos main_codifica.c e main_decodifica.c, contendo o código-fonte de programas para testar as funções pedidas.

2) Arquivos imagem.c e imagem.h, que implementam um subconjunto do formato BMP, usado para ler e apresentar as imagens de teste. Você não precisa compreender o conteúdo destes arquivos.

3) Um arquivo trabalho1.h, contendo as declarações das funções pedidas e de funções auxiliares. Este arquivo deve ser incluído no seu arquivo .c através da diretiva `#include`. As funções declaradas no arquivo se dividem em 3 grupos:

- Funções do trabalho: estas são as funções pedidas, que cada um deve implementar.

- Funções auxiliares que devem ser chamadas pelos alunos: estas são as funções mencionadas na descrição das funções pedidas. Você deve fazer uso destas funções.

- Funções auxiliares que não devem ser chamadas pelos alunos: estas funções são usadas nos testes, mas não devem ser usadas no seu trabalho.

3) Um arquivo trabalho1.c, contendo as implementações das funções auxiliares, assim como dados internos usados para os testes. Você não precisa compreender o conteúdo deste arquivo, nem mesmo das funções auxiliares usadas pelo seu trabalho. Lembre-se: é preciso abstrair!

4) Um diretório img, contendo imagens de teste no formato BMP e imagens codificadas, para fins de teste.

Para usar os arquivos disponibilizados:

1) Crie no Code::Blocks um projeto vazio (Empty project).

2) Coloque todos os arquivos disponibilizados no mesmo diretório criado para o projeto.

3) Adicione ao projeto os arquivos imagem.c, imagem.h, trabalho1.c, trabalho1.h, e um dos arquivos main_.c.

4) Crie o arquivo para as suas funções, e adicione-o ao projeto.

5) Crie corpos "vazios" para as funções pedidas. Os protótipos podem ser copiados/colados diretamente do arquivo trabalho1.h.

6) Modifique os nomes dos arquivos de entrada e saída (`ARQUIVO_IMG` e `ARQUIVO_RBD`) no arquivo principal.

7) Verifique se o projeto pode ser compilado e o programa executado.

A compressão de dados é uma área tradicional da computação, de grande importância e utilidade prática. Métodos de compressão podem reduzir consideravelmente a quantidade de bytes necessários para representar determinado conjunto de dados. Várias estratégias foram propostas para compressão, tanto para dados genéricos – por exemplo, formatos como ZIP e RAR – quanto para dados de tipos específicos – por exemplo, formatos como MPEG e JPG. A compressão pode ser sem perdas ou com aproximações, que sacrificam parte do conteúdo original em troca de taxas maiores de compressão.

Tomemos como exemplo uma imagem no formato conhecido como *Full HD*, ou 1080p. Esta imagem tem resolução de 1920x1080 pixels, ou seja, é formada por um total de 2.073.600 pontos. Cada ponto da imagem possui uma cor, representada por 3 bytes. Ou seja, uma única imagem possui 6.220.800 bytes (em torno de 6MB). Ignorando quaisquer outros dados, uma sequência de apenas 10 minutos de vídeo *Full HD,* a 30 quadros por segundo (FPS) e sem compressão, ocuparia 111.974.400.000 bytes – ou seja, mais de 100GB!!! Comparilhamento de vídeos e serviços de *streaming* seriam inviabilizados se não existissem técnicas para compressão de dados.

Neste trabalho, você deve implementar um esquema simples de compressão, que chamaremos de "RBD" (*Reduced Bit Depth*). Por simplicidade, consideraremos que as imagens de entrada possuem apenas tons de cinza. Nesta representação, chamada de escala de cinza, ou *grayscale*, cada ponto que forma a imagem é dado em um único byte. Um pixel com valor 0 é preto, um pixel com valor 255 é branco, e valores intermediários representam diferentes tons de cinza - quanto maior o valor, mais claro o pixel. Nosso esquema de compressão deve reduzir o número de bits usado para cada pixel (*bit depth*). Para isso, devem ser descartados sempre os bits menos significativos de cada pixel.

Por exemplo, suponha uma série de 8 pixels, com os valores 0x12, 0xAF, 0x78, 0xBC, 0x33, 0x51 0x49 e 0xD1. Colocando esses valores lado a lado, temos a sequência de 8 bytes abaixo:

12 AF 78 BC 33 51 49 D1

Se em vez de 8 bits usarmos 2 bits para representar cada pixel, os 6 bits menos significativos de cada byte seriam descartados, os bits seriam reagrupados, e a sequência ficaria com apenas 2 bytes:

26 17

Para descomprimir uma sequência comprimida, cada grupo de *n* bits (onde *n* indica o *bit depth* da imagem comprimida), deve ser reexpandido para 8 bits. Os bits que haviam sido descartados devem ser substituídos por zeros. Desta forma, os 2 bytes do exemplo anterior seriam expandidos novamente para 8 bytes:

00 80 40 80 00 40 40 C0

Se tomarmos novamente a mesma sequência inicial, mas usando 4 bits para representar cada pixel, a compresão seria para 4 bytes:

1A 7B 35 4D

A versão descomprimida desta sequência seria:

10 A0 70 B0 30 50 40 D0

De certa forma, o que a compressão RBD faz é simplesmente arredondar os bytes, igualando todos os valores que diferem somente nos bits menos significativos – o nome técnico para isto é “quantização”. Por exemplo, se o *bit depth* da imagem comprimida for de 4 bits, todos os pixels com valores entre 0xA0 e 0xAF seriam comprimidos para os mesmos 4 bits, e receberiam o valor 0xA0 ao serem expandidos. A quantização de dados é um dos principais conceitos explorados por técnicas de compressão usadas na prática – outro princípio importante, que é a redundância dos dados, não será explorado neste trabalho. É claro, os algoritmos mais utilizados na prática, como o JPEG, utilizam esquemas muito mais sofisticados do que este!

Apesar de simples, a técnica proposta para compressão é capaz de reduzir o espaço ocupado por uma imagem. Por exemplo, uma imagem em escala de cinza com 2048x1536 pixels, com 1 byte por pixel, exige 3.145.728 bytes somente para os dados. Com o *bit depth* reduzido para 4 bits por pixel, este número seria reduzido pela metade, para 1.572.864 bytes. Esquemas mais sofisticados seriam capazes de comprimir a imagem com taxas consideravelmente maiores de redução, e com melhor qualidade.

O seu objetivo neste trabalho é criar funções para codificação e decodificação de imagens usando o esquema descrito acima. Por simplicidade, só serão considerados válidos os *bit depths* 1, 2 e 4. Entretanto, construa a sua solução de forma o mais genérica possível, evitando de criar 3 blocos separados que cubram, cada um, um dos casos. Como desafio adicional, consideraremos que os dados são enviados e recebidos em *streams* seriais que trabalham com apenas 1 byte por vez – imagine um esquema de comunicação bastante limitado, como LEDs que piscam. Isso quer dizer que, ao codificar uma imagem, você não pode acessar uma posição arbitrária dentro dela – pode apenas pedir pelo “próximo pixel”. Da mesma forma, ao decodificar uma sequência comprimida, você só pode pedir pelo “próximo byte”. No total, você deve criar duas funções:

Função 1 (45 pontos):

```
void codificaStreamImagem (int n_bits);
```

Esta função deve ler uma imagem a partir de um *stream*, e enviar a outro *stream* os bytes da imagem comprimida, com o *bit depth* reduzido para `n_bits`. O valor de `n_bits` precisa ser entre 1, 2 ou 4 – se isso não ocorrer, a função deve retornar sem realizar nenhuma ação. Para controlar os *streams*, você deve usar duas funções, com os protótipos abaixo:

```
unsigned int pegaProximoPixel ();
void enviaByteRBD (unsigned char byte);
```

As duas funções estão declaradas no arquivo trabalho1.h e implementadas no arquivo trabalho1.c. Você não precisa escrever estas funções, nem sequer compreendê-las – basta usá-las e elas simularão um *stream* de entrada para a imagem e um *stream* de saída para os bytes da imagem comprimida. Você também não precisa se preocupar com a disposição dos pixels em linhas e colunas – simplesmente suponha que os pixels da imagem e os bytes da imagem comprimida estão todos em sequência.

A função `pegaProximoPixel` retorna um número entre 0 e 0xFF, que indica a intensidade do próximo pixel da imagem. Se o stream tiver chegado ao fim, a função retorna o valor especial 0xFFFFFFFF.

A função `enviaByteRBD` recebe como parâmetro o próximo byte que deve ser enviado ao stream de saída.

Por exemplo, suponha que nossa imagem possui apenas 4 pixels, com os valores 0x12, 0xAF, 0x78 e 0xBC. Neste caso, a função `pegaProximoPixel` será chamada 5 vezes - mas este número não é conhecido até que a função retorne 0xFFFFFFFF. Os 4 bytes válidos recebidos como entrada seriam "empacotados" dependendo do valor de `n_bits`. Se `n_bits` for 4, a função `enviaByteRBD` deve ser chamada 2 vezes, para os bytes 0x1A e 0x7B. Note que, como não podemos usar vetores ou matrizes para armazenar a sequência completa, precisamos enviar os bytes para a saída tão logo tenhamos eles disponíveis.

Note que existe a possibilidade de termos bytes "incompletos" ao final da imagem – por exemplo, se a sequência original tivesse 3 bytes, uma compressão com `n_bits` igual 4 completaria 1 byte e meio – o primeiro byte de saída armazenaria os 2 bytes iniciais da imagem, e o byte incompleto ficaria com o terceiro byte da imagem. Nestes casos, o conteúdo dos bits menos significativos do último byte enviado serão ignorados pela função `enviaByteRBD`.

Função 2 (50 pontos):

```
void decodificaStreamRBD (int n_bits, int preenche);
```

Assim como a função `codificaStreamImagem` comprime uma imagem recebida como entrada, a função `decodificaStreamRBD` deve decodificar bytes recebidos de um *stream*, enviando a outro *stream* os valores dos pixels de uma imagem. Para isso, você deve usar duas funções, com os protótipos abaixo:

```
unsigned int pegaProximoByteRBD ();
void enviaPixel (unsigned char pixel);
```

A função `pegaProximoByteRBD` retorna um grupo de 8 bits, que devem ser usados para "remontar" os pixels da imagem. Quando não existirem mais bytes disponíveis, a função retorna o valor especial 0xFFFFFFFF.

A função `enviaPixel` recebe como parâmetro o próximo pixel reconstruído que deve ser enviado ao *stream* de saída.

A relação entre a quantidade de bytes recebidos e os pixels produzidos na saída é dada pelo parâmetro `n_bits`. Por exemplo, se o primeiro byte recebido na entrada for 0xFE e `n_bits` for 4, devemos enviar à saída dois pixels: 0xF0 e 0xE0. Para `n_bits` igual a 2, o mesmo byte 0xFE geraria 4 pixels, com os 3 primeiros valendo 0xC0 e o quarto valendo 0x80. Já para `n_bits` igual a 1, o mesmo byte 0xFE geraria 8 pixels, todos com valor igual a 0x80, exceto o último, que valeria 0x00.

Desafio Extra (5 pontos):

O parâmetro `preenche` da função `decodificaStreamRBD` é uma flag. Se ela for verdadeira, os bits menos significativos dos pixels reconstruídos devem ter valores aleatórios, em vez de zero. Para receber a pontuação extra, além de implementar esta funcionalidade, você deve explicar no seu relatório:

a) Qual o efeito do preenchimento com zeros no aspecto das imagens reconstruídas?
b) Qual o efeito do preenchimento com zeros na intensidade média das imagens reconstruídas?
c) Qual o efeito do preenchimento com valores aleatórios no aspecto das imagens reconstruídas?
d) Qual o efeito do preenchimento com valores aleatórios na intensidade média das imagens reconstruídas?

Nota: você *pode* (e *deve*!) usar funções da biblioteca-padrão para implementar esta (e somente esta) funcionalidade.