# Trabalho Prático de AEDS 3
# Algoritmo de Compressão LZ77

*Nome: Ronald Davi Rodrigues Pereira*
*Turma: TD1*
*Matrícula: 2015004437*

## Introdução

Atualmente, a compressão de dados é indispensável para a transmissão de dados. O Youtube, Google Drive, Dropbox, entre outros, utilizam algorítmos de compressão para que seus trabalhos sejam sempre o mais otimizado possível.

A compressão de dados se dá pelo fato de que um vídeo no Youtube, por exemplo, demoraria muito mais tempo do que leva atualmente. A taxa de FPS (frames per second) de um vídeo de alta resolução (HD) chega a ser enorme por causa de sua pixelagem alta. Portanto, um vídeo de 5 minutos que é carregado atualmente em menos de 2 minutos, demoraria, caso não houvesse a compressão de dados, mais do que 1 hora para ser processado e exibido.

Desse modo, percebe-se a importância de Algoritmos de Compressão de dados, de modo a fomentar o objetivo desse trabalho: Implementar um algoritmo de compressão e descompressão de quaisquer dados.

## Solução do Problema

O algoritmo pensado para o programa compress.exe foi o seguinte:

Dada uma entrada "abaabadadadadac" (escrita em um .txt), o programa analisa as redundância de cadeias de chars maiores ou iguais à 3 e as referencia por um ponteiro, onde o primeiro número referencia o comprimento da string redundante e o segundo número referencia a distância do primeiro char redundante (andando para trás) em chars. Desse modo, a saída didática do programa seria:

**aba<3,3>da<6,2>c**

Dado essa saída, o programa converte cada char em inteiro através de um casting ((int)char) e os registra novamente. Ficando assim:

**97 98 97 <3,3> 100 97 <6,2> 99** (os espaços foram inseridos para simplificar o entendimento)

Desse modo, o programa finaliza a sua execução transformando esse registro em número binários. Utilizando o bit menos significativo para identificar um ponteiro (1) ou um literal (1).

**00110000 10011000 10001100 00110000 00000000 00000000 01000110 01000011 00001100 00001100 00000000 00001001 10001100 00010100**

O algoritmo pensado para o programa decompress.exe foi o seguinte:

Dada uma entrada comprimida (saída do compress.exe), o algoritmo de descompressão faz os passos contrários que o algoritmo de compressão fez. Basicamente, ele transforma cada registro em forma de bits em números literais e ponteiros novamente. Seguindo na mesma linha do exemplo acima, a saída seria:

**97 98 97 <3,3> 100 97 <6,2> 99** (os espaços foram inseridos para simplificar o entendimento)

Desse modo, ao se fazer um casting de número para char novamente, usando a função atoi(), tem-se o resultado abaixo:

**aba<3,3>da<6,2>c**

Com isso, o processamento reverso ficou trivial. Portanto, a transformação faltante é apenas a dos ponteiros referenciais às redundâncias. Desse modo, ao se fazer a transformação, o resultado é:

**abaabadadadadac**

O que possibilita voltar ao estado incial da entrada do compress.exe.

## Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código está dividido em 6 arquivos principais e um arquivo em linguagem bash para compilação: compress.c, function_compress.c e function_compress.h constituem o primeiro programa (compress.exe). decompress.c, function_decompress.c e function_decompress.h constituem o segundo programa (decompress.exe). Além disso, um arquivo Makefile contem a programação em bash para a compilação dos arquivos de ambos os programas.

O compilador utilizado foi o GCC versão 4.9.5 no sistema operacional Linux Ubuntu 14.04. Para executá-lo basta digitar no Terminal os comandos (tendo em mente que o arquivo de entrada e os arquivos .c e .h terão que estar contidos no mesmo diretório que o programa principal):

```
make all
./compress.exe nomeDaEntrada nomeDaSaída
./decompress.exe nomeDaEntrada nomeDaSaída
make clean
```

## Análise de complexidade:

A análise de complexidade será feita em função da variável *n*, que representa o número de operações relevantes dadas (atribuições) das principais funções do programa compress.exe.

int BMH(): De acordo com o livro Projeto de Algoritmos, de Nivio Ziviani, a complexidade dessa função é de O(n*m) no pior caso e O(n/m) no melhor caso, sendo m o tamanho da minha string maior a ser comparada.

void conversaoEmBinarioLiteral(): O(1) por conter apenas um loop que é executado um número fixo de vezes.

void conversaoEmBinarioPonteiroComprimento(): O(1) por apenas fazer a chamada de uma função.

void conversaoEmBinarioPonteiroR_Offset(): O(1) por conter apenas um loop que é executado um número fixo de vezes.

void escritaEmBinario(): O(n) por conter um loop que irá fazer várias operações, porém menor que n vezes (evitando $O(n^2)$).

void matching(): O(1) por apenas fazer a impressão em um arquivo.

void noMatching():O(1) por apenas fazer a impressão em um arquivo.

void compression(): O(n) por conter um loop que irá fazer várias operações, porém menor que n vezes (evitando $O(n^2)$).

CHAR *doJumps(): O(1) por apenas fazer operações dentro de um loop que tem determinado número fixo de execuções.

void fazLeitura(): O(n) por fazer operações dentro de um loop que tem determinado tamanho igual ao tamanho da entrada.

void formataBinario(): O(n) por fazer operações dentro de alguns loops (os demais tem custo de O(1)) que tem determinado tamanho igual ao tamanho da entrada.

A análise de complexidade será feita em função da variável *n*, que representa o número de operações relevantes dadas (atribuições) das principais funções do programa decompress.exe.

void transformaPonteiroR_Offset(): O(1) por conter um loop que é executado um número fixo de vezes.

void transformaPonteiroComp():O(1) por conter um loop que é executado um número fixo de vezes.

void transformaLiteral():O(1) por conter um loop que é executado um número fixo de vezes.

void transformaEmNumeros(): O(n) por conter um loop principal que é executado n vezes, sendo n o tamanho da entrada do programa. Mesmo contendo

outros loops aninhados, a sua complexidade não aumenta, pois todos os loops possuem custo de O(1), por serem executados um número fixo de vezes.

void imprimePonteiroNormal():O(1) por conter um loop que é executado um número fixo de vezes.

void imprimePonteiroRepete(): O(n) por conter um loop que é executado n vezes e um loop interior que é executado um número fixo de vezes.

void montaPonteiro(): O(1) por somente conter chamadas de funções.

void descomprime(): O(n) por conter um loop que é executado n vezes, sendo n o tamanho da entrada do programa.

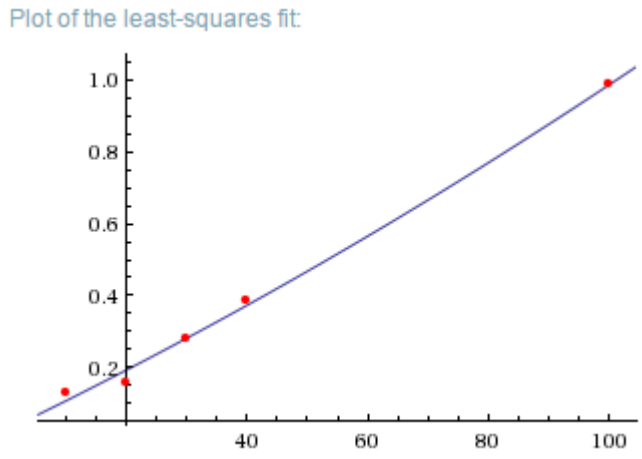
## Avaliação experimental:

Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. O teste que será utilizado abaixo foi realizado em um Intel Core i3, com 4 Gb de memória RAM.

Foram analisados entradas de tamanhos diferentes(n = 10 chars), variando apenas os números de multiplicador das entradas (n , 2n, 3n, ..., 10n).

n → 0.008 segundos
2n → 0.006 segundos
3n → 0.013 segundos
4n → 0.017 segundos
[...]
10n → 0.034 segundos

A partir desses pontos de coordenadas foi gerado o gráfico de regressão quadrática abaixo:




Função quadrática que melhor expressa a função de tempo do programa:
$0.0000151626x^2 * 0.00809114x + 0.0245317$

Com isso, pode-se concluir que o tempo vai aumentando quase de maneira linear, comprovando o custo do programa ser O(n), ao se considerar uma taxa de erro para medição dos tempos de execução.

## Conclusões:

Diferentes tipos de arquivos recebem diferentes taxas de compressão pelo fato da necessidade de tamanho para representar um arquivo. Por exemplo, um arquivo do tipo jpg , na maioria dos casos, necessita de mais espaço para a sua representação do que um arquivo do tipo txt de 80 linhas. Sendo assim, a diferenciação dos tipos de arquivos leva, na maioria das vezes, à diferenciação da taxa de compressão.

Há diferenças da taxa de compressão de arquivos, que dependem do tipo de arquivo a ser comprimido. Isso se dá pelo fato do tamanho do arquivo a ser processado. Por exemplo, uma imagem possui um arquivo maior que um arquivo de texto simples, logo seu tamanho será maior. Porém, a taxa de compressão é algo muito situacional, pois depende da disposição dos caracteres de maneira redundante, mas também, o tamanho do arquivo pode aumentar a possibilidade de se ter essas redundâncias.

Sendo assim, quanto maior o tamanho do arquivo de entrada, maior a possibilidade de maior taxa de compressão, pois é maior a possibilidade de se possuir redundâncias.

## Anexos:

**Listagem dos programas:**

- **compress.c (Programa Principal Compress)**
- **decompress.c (Programa Principal Decompress)**
- **function_compress.c e function_compress.h (Library contendo as funções do Compress)**
- **function_decompress.c e function_decompress.h (Library contendo as funções do Decompress)**
- **Makefile (Arquivo em bash para compilação dos programas)**