**Anticompressor**

**João Vitor Boer Abitante**

Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

Porto Alegre – RS – Brasil

Joao.Abitante@edu.pucrs.br

***Resumo:*** *Este artigo apresenta a solução para calcular a quantidade de caracteres da string de um documento pós processo de anti-compressão, aonde no documento, há letras, e cada letra tem ou não, sua respectiva frase. Será apresentada a descrição do problema, sua modelagem, uma solução e pseudocódigos do algoritmo criado. Para concluir são testados 10 casos fornecidos e é feita a análise dos resultados obtidos. O respectivo artigo compõe a disciplina de Algoritmos e Estrutura de Dados II, da Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul.*

# Introdução

O problema a ser investigado pode ser resumido assim: temos dados comprimidos em um arquivo (Figura 1), que podem ser facilmente entendidos como um dicionário, onde o *char* é a chave e a *string* é o valor.

Table

Description automatically generated

Figura 1. Caso informado no enunciado do trabalho.

Deve-se então, descomprimir e encontrar a a quantidade de letras da *string* final, ou seja, sempre que encontrarmos a letra *“u”* devemos transformá-la em *“mimimi”*. Visando solucionar o problema, seguem-se três principais regras:

1. Encontrar o *char* e a *string* aonde inicia-se a substituição.

2. Tratar corretamente as chaves que não contém valor, por exemplo, a letra *“m”* da Figura 1.

3. A descompressão é encerrada somente quando a *string* final não tiver nenhum *char* para ser substituído.

Por exemplo, seguindo estas regras obtemos como resultado:

* Letra Inicial: *“a”*
* *String* final:



Figura 2. Resultado do caso do enunciado do trabalho.

O problema a ser resolvido é determinar a quantidade final de carateres da *string* e o tempo que levou para executar o programa. Segundo o enunciado do trabalho, este caso teria um resultado total de 47 caracteres na *string* pós processo de anticompressão. Ao todo, são 10 arquivos de casos de teste fornecidos pelo professor que devemos descomprimir.

A fim de solucionar este problema, analisaremos uma solução possível que seja eficiente. Após, os resultados obtidos serão apresentados em uma tabela juntamente com as conclusões realizadas.

# Modelagem

Ao analisar o que foi solicitado podemos pensar em duas opções: fazer as substituições e no final calcular o tamanho da *string* resultante ou guardar em uma variável o tamanho da *string* substituta sempre que for possível substituir. Tendo em vista essas duas possibilidades e a performance do código, escolhemos a segunda opção, pois na primeira seria necessário realizar uma quantidade enorme de substituições que resultariam num custo de processamento imenso.

Considerando-se a escolha acima e a maneira como os casos estão escritos no arquivo, optamos por armazenar as informações em uma estrutura de lista (desconsiderando os *chars* em que não há respectiva *string)* onde cada posição será um objeto de uma classe chamada Nodo, que contém os seguintes atributos: “sozinha”, “sequência”, “primeiro” e “tamanho”. Estes correspondem respectivamente ao *char*, a *string*, um *boolean* que começará em True, e o tamanho em que a *string* vai adquirindo até não poder mais se operar sobre ela. Porém este último começará em 0.

class Nodo:

def construtor (sozinha, sequencia, primeiro, tamanho):

sozinha = sozinha

sequência = sequencia

primeiro = primeiro

tamanho = tamanho

Logo, nossa lista ficaria assim:

|  |  |  |  |
| --- | --- | --- | --- |
| Nodo 1:  a  memimomu  True  0 | Nodo 2:  e  mimomu  False  0 | Nodo 3:  i  mooo  False  0 | Nodo 4:  u  mimimi  False  0 |

Figura 3. Exemplo da lista de classes.

Os métodos implementados para encontrar a solução para cada um dos casos fornecidos serão descritos a seguir.

# Solução

O primeiro passo para resolver o desafio é realizar a leitura dos arquivos de casos de teste colocando os valores corretos na lista de classes. Para começar, precisamos ler as linhas e separar a letra sozinha de sua sequência. Com está constatação, podemos propor uma resposta bastante simples: visto que cada linha é um vetor de *chars*, bastaria pegar a posição *linha[0]* para ser a letra inicial e consequentemente pegar o que sobrou (ignorando o espaço) para ser a frase correspondente (*linha[2:-1]).*

Porém, como foi citado na introdução, faz-se necessário desconsiderar os *chars* onde não há respectiva *string.* E para isso, utiliza-se de uma condição que verifica a existencia da mesma. Caso a condição for verdadeira, o algoritmo adiciona um objeto na lista contendo a letra inicial encontrada, sua frase, um parâmetro True (será explicado posteriormente), e o número inteiro 0. Um algoritmo implementando está idéia seria parecido com este:

abrir (‘casox’, ‘ler’) as arquivo:

para linha em arquivo:

if linha[2] != ‘\n’:

sozinha = linha[0]

sequencia = linha[2:-1]

lista\_classes.adicionar(Nodo(sozinha,sequencia, True,0))

Após ter lido o arquivo e organizado corretamente as informações, é hora de encontrar qual é a letra incial do processo de descompressão. Para isso seguimos a ideia de que a letra não pode estar contida em nenhuma das sequencias do arquivo. Sabendo disso, podemos pensar em uma alternativa: realizar a verificação de cada letra com todas as sequências obtidas. Se essa verificação for válida, ou seja, a letra está em alguma das sequências, alteramos o parâmetro *primeiro* para False, caso contrário, permanecerá em True. Esta solução seria assim:

para i < tamanho(lista):

para j linha tamanho(lista):

if lista[i].sozinha in lista[j].sequencia:

lista[i].primeiro = False

Com a letra inicial encontrada o programa está pronto achar o resultado. Então, passamos como parâmetro para uma função cada *char* da sequência que corresponde a letra inicial e a lista contendo as classes. Seguindo o exemplo e resultado baseado na Figura 1, teríamos algo deste tipo:

Letra inicial: “*a”*  m e m i m o m u

função (m, lista)

Esta função se baseia em uma série de recursões, que descobrem e retornam a quantidade de caracteres que cada letra recebida por parâmetro gera. Para isso, é executado uma comparação afim de encontrar as informações do objeto que contém a letra recebida, porém, se ele não for encontrado, isso significa que o tamanho de sua *string* é 1, já que não foi registrado na lista pois não tem uma respectiva sequência. Logo, temos que retornar este valor. Entretanto, se o objeto for localizado temos 2 caminhos para retornar a soma: caso o seu tamanho ainda seja igual a 0, chamamos a recursão da função, que atualizará o atributo tamanho do objeto baseado na soma da quantidade total *chars* das *strings* que seriam substituídas. Se for diferente de 0, significa que o programa já sabe o tamanho máximo que cada objeto tem, logo, atribui-se este valor a variável soma. Este processo descrito por ser melhor compreendido analisando o código abaixo.

def funcao(vetor, letra):

soma = 0

encontrou = False

para i < tamanho(vetor):

if vetor[i] == letra:

para j in vetor[i].sequencia:

encontrou = True

soma += Soma(vetor, j)

vetor[i].tamanho = soma

break

else:

encontrou = True:

soma = vetor[i].tamanho

if encontrou == False:

soma = 1

retorna soma

Considerando-se que submetamos o caso da Figura 1 nesta função, seus retornos seriam iguais os da tabela abaixo. Se efetuarmos a soma da segunda coluna temos a confirmação que o resultado encontrado pela função está correto, visto que os dois resultam em 47.

|  |  |
| --- | --- |
| Letra | Quantidade de caracteres gerados |
| m | 1 |
| e | 23 |
| m | 1 |
| i | 4 |
| m | 1 |
| o | 1 |
| m | 1 |
| u | 15 |

Tabela 1. Exemplos da quantidade de caracteres.

Se somarmos a segunda coluna, obtemos o número 47, que é o resultado correto.