# O anticompressor de textos

Mariela Pontes Cordeiro[*], Victor dos Santos Cruz[†]
Faculdade de Informática — PUCRS

19 de abril de 2022

**Resumo**

Este relatório descreve nossa solução para o problema proposto no primeiro trabalho da disciplina de Algoritimos e Estruturas de Dados II, no qual deve-se desenvolver um algoritmo anticompressor de textos.

**Palavras-chaves**: Algoritmos. Estrutura de Dados. Anticompressão.

# Introdução

Dentro do escopo da Engenharia de Software, um algoritmo é uma sequência finita de ações que tem como objetivo obter uma solução para um problema. Neste trabalho, foi solicitado o desenvolvimento de um anticompressor de texto que funciona através da transformação de uma letra do alfabeto em uma sequência de letras associada à mesma, e cada uma das letras dessa sequência pode possuir, ou não, sua respectiva sequência de letras associada, iniciando um processo de substituição que deve ser o mais eficiente possível e que deve terminar quando mais nenhuma substituição pode ser realizada.

Considerando o problema em questão, foram disponibilizadas dez tabelas de substituição para serem testadas que podem ser exemplificadas pela tabela abaixo:

```
a memimomu
e mimomu
i mooo
u mimimi
o
m
```

Para iniciar o desenvolvimento, deve-se em primeiro lugar descobrir qual a letra que vai iniciar todas as substituições; ela não será a primeira na ordem lida, pois é necessário encontrar uma letra que apareça apenas uma vez no arquivo e que não esteja em nenhuma sequência de letras, mas que possua uma associada; o motivo dessa lógica é que se a primeira letra pudesse estar no lado direito, o algorítmo entraria em um *loop* infinito, e, também, o anticompressor necessita de uma sequência associada à letra para iniciar a construção do arquivo final anticomprimido.

Assim, depois que a primeira letra é encontrada e todas as substituições forem realizadas, deve ser informado quantas letras resultaram da anticompressão e o tempo em que cada caso de teste foi concluído.

[*] mariela.c@edu.pucrs.br
[†] v.cruz001@edu.pucrs.br

# Encontrando a letra inicial

Inicialmente, tivemos que analisar o problema por algum tempo até entender a lógica da letra que dará início às substituições, e, depois disso, decidimos que fazer o uso da coleção *Set()* na linguagem de programação Python seria apropriado para guardar todas as letras que constituem as substituições por inúmeros motivos: nesse tipo de coleção a ordem das letras não importa, não há letras repetidas e a maior vantagem é que o método para conferir se um certo elemento está contido no *set* é muito otimizado.

Nossa solução utiliza quatro coleções diferentes para essa busca, uma lista, uma lista temporária, um dicionário e um set. A grosso modo, a ideia é ler cada linha do arquivo e guardar a letra e a sequência de letras associada dentro de uma lista temporária com duas posições, e, durante cada leitura, é conferido se o tamanho dessa lista é menor do que dois, porque, caso for, significa que a letra não possui associação, logo não é preciso guardá-la. Ao contrário das letras sem substituição, as que possuem estão armazenadas na lista temporária que é dividida resultando em duas *strings*, uma com a letra e a outra com a sequência de letras, iremos chamá-las respectivamente de k (letra) e de v (sequência) para melhor compreensão.

Após essa divisão, inserimos as *strings* dentro da outra lista onde cada posição irá conter tanto o k quanto o v alocados, e inserimos cada v dentro do nosso set utilizando o operador de *unpacking* para separá-las.

```
1  função anticompressor(arquivo)
2  substitution_table ← {} # Dicionário vazio
3  substitution_table_list = [] ← # Lista vazia
4  all_leters ← set() # Conjunto/set vazio (dicionário chave:chave)
5  com arquivo begin
6    para linha em arquivo begin
7      k_v ← linha separada em espaços
8      se k_v tiver menos que dois itens
9        continue
10     k , v ← k_v # Separa a lista em chave e valor
11     substitution_table[i] ← k , v
12     all_letters ← soma com todas as letras em v
13   fim
14 fim
```

Obtendo todos os dados necessários armazenados nas estruturas já comentadas, criamos uma *string* inicialmente vazia que irá armazenar a palavra inicial quando encontrada; e, para isso, iteramos pela lista em que cada posição possui o k e o v ao passo que inserindo cada um como chave e valor no dicionário já instanciado, ou seja, temos agora um dicionário em que a chave é a letra e o valor é a sequência de letras associada; em seguida realizamos a verificação em que se a *string* da letra inicial ainda estiver vazia e se o k (letra) não estiver contido no *set* que possue todas as letras de v não ordenadas e não repetidas, então o k será a letra inicial.

```
1  first_letter ← ""
2  para chave , valor em substitution_table_list begin
3    substitution_table[k] ← v
4    se first_letter for vazia e valor não estiver em all_letters
5      first_letter ← chave
6  fim
```

Quando a primeira letra é encontrada a iteração acaba e tanto o *set* quanto a lista são deletadas para liberar memória e tornar a execução ainda mais rápida. Desse modo, o primeiro problema apresentado é resolvido pelo nosso algoritmo e os resultados serão mostrados um

pouco mais abaixo.

# Calculando o tamanho final do arquivo anticomprimido

Em um primeiro momento, nossa solução retornaria a *string* final sem utilizar recursão, o que torna o processo extremamente mais custoso quando comparado ao da solução final que desenvolvemos, em que não é necessário retornar o arquivo final com as letras, mas somente quantas vezes seria necessário substituir cada letra, somando o valor dela que estará armazenado para que ele seja apenas calculado uma vez. De qualquer forma, achamos pertinente mostrar as duas soluções considerando que a primeira foi essencial para que pudéssemos chegar na segunda.

## 0.1 Primeira solução

A solução acaba por ser relativamente simples, e posiciona-se dentro do mesmo método que encontra a letra inicial; utilizando o dicionário que foi populado com todas as letras que possuem palavras associadas, alocamos uma *string* que possui a substituição da letra inicial e também um *set* que guarda todas as chaves desse dicionário (letras).

Enquando a última estrutura mencionada for maior do que zero, é realizada uma intersecção entre a mesma e cada letra da palavra que dará início à construção final, o que resulta em uma coleção que possui apenas as letras que deverão ser substituídas naquela específica sequência. Se o tamanho da intersecção for zero, a iteração acaba, caso contrário, para cada letra dela é realizada uma substituição em cima da *string* trocando todas as letras necessárias por suas respectivas sequências de letras.

Assim, a iteração acaba quando não houver mais nenhuma letra que pode ser substituída, e é retornado o tamanho dessa string, além da primeira letra. A solução será mostrada abaixo:

```
1  res ← substitution_table[first_letter] # String com a sequência de letras
2  keys ← set(substitution_table.keys()) # Conjunto/set de chaves
3  enquanto set > 0
4      subst_keys ← keys & set(res) # Interesecção
5      se subst_keys for igual a 0
6          fim
7      para chave em subst_keys
8          res ← res.substitui(chave , sequência da chave)
9  fim
10 retorna tamanho de res e first_letter
```

Considerando que o problema proposto não pede o arquivo final, mas sim o tamanho dele, é visível que não é necessário todo esse trabalho, até porque, no final, o que iremos retornar é apenas o tamanho da *string* anticomprimida. Sendo executado em nossas máquinas, ele não passa do segundo caso, acreditamos que o terceiro caso já levaria algumas horas; esse algoritmo cresce de forma exponencial. O tempo utilizado para os dois casos que foram executados são:

| | | | |
|---|---|---|---|
| caso01.txt | tempo = 0.01099896 | tamanho = 178833 | letra = q |
| caso02.txt | tempo = 31.9802839 | tamanho = 399461207 | letra = i |

## 0.2 Segunda solução

Essa é a solução final do nosso trabalho, pois ela é inúmeras vezes mais eficiente do que a outra para entregar o resultado pedido. A lógica dela é utilizar um método recursivo para calcular quanto vale passar por cada letra, ou seja, pela palavra associada, usando o valor

um para se caso a letra não tiver nenhuma associação. Se a letra já tiver um valor salvo no dicionário de valores criado para armazenar os valores das letras com um inteiro, é retornado esse valor, caso contrário é passado por cada letra da palavra sendo analisada no momento, calculando o valor dela observando sua substituição. O dicionário de valores do caso de exemplo apresentado na introdução seria:

```
a 47
e 23
i 4
u 15
o 1
m 1
```

Acreditamos que cada passo dessa parte do algorítmo pode ser explicada mais profundamente após o pseudocódigo ser mostrado.

```
1  value_cache ← {} # Dicionário vazio
2  função calc_entry_value(entrada: str) → int
3      total_val ← 0
4      se entrada in value_cache
5          retorna value_cache[entrada]
6      entrada ← substitution_table.get(entrada , entrada) # Sequência associada
7      para char in entrada
8          se char ← entrada
9              retorna 1
10         value ← calc_entry_value(char)
11         value_cache[char] ← val
12         total_val += val
13     fim
14     retorna total_val
15 retorna calc_entry_value(first_letter) , first_letter
```

De acordo com esse trecho, é criado um dicionário que irá armazear o valor em inteiro de custo de cada letra, após isso definimos que a entrada é uma string, mas a saída será um inteiro. Uma variável irá ser usada para ir somando o valor de cada letra quantas vezes ela for usada nas substuições para que assim possa ser calculado o valor final da quantidade de letras do arquivo.

1. se a letra - na primeira vez será a letra inicial - já estiver salva no dicionário com um valor, é retornado ele, caso contrário a entrada será a susbtituição da letra.

2. para cada caractere na substituição, se ele não tiver nenhuma associação, ele será igual à entrada, logo o valor um é retornado.

3. em todos os outros casos, o método utiliza a recursão para analisar e calcular o valor da letra; ao longo do preenchimento do dicionário é somado o valor de cada letra e, finalmente, é retornado o tamanho do arquivo final.

### 0.2.1 Calculando o tempo de execução

Para calcular o tempo de execução do algoritmo e também receber os resultados do método anticompressor, optamos por executar mil vezes o processo de iterar por cada caso de teste a fim de obter resultados mais precisos. Dessa forma, utilizamos dois dicionários, um que irá

armazenar o caso de teste e o tempo levado, já o outro o tamanho do arquivo final e a primeira letra.

```
exec_times ← {} # Dicionário vazio
results ← {} # Dicionário vazio
enquanto < 1000
    para arquivo em diretório 'casos' begin
        se arquivo not em exec_times:
            exec_times[file] ← []
        start_time ← time.time()
        res , first_letter ← anticompressor(arquivo)
        exec_times[arquivo] recebe time.time() −start_time
        se arquivo not em results:
            results[arquivo] ← (res , first_letter)
    fim
fim
para f, t em itens de exec_times
    _time ← soma(t) / tamanho(t)
    exec_times[f] ← _time
fim
```

Nesse trecho temos três iterações diferentes; a primeira é apenas para obter mais precisão no tempo de execução, pois alguns casos foram tão rápidos que retornavam 0.0. O segundo é responsável por ler todos os casos no diretório "casos". No terceiro *loop*, é feita a média dos tempos, onde é somado todos os tempos das execuções de cada caso e dividido pelo tamanho, que nesse caso será mil. Após esse processo utilizamos a função *print()* com iteração para obter os conteúdos dos dicionários.

# Resultados

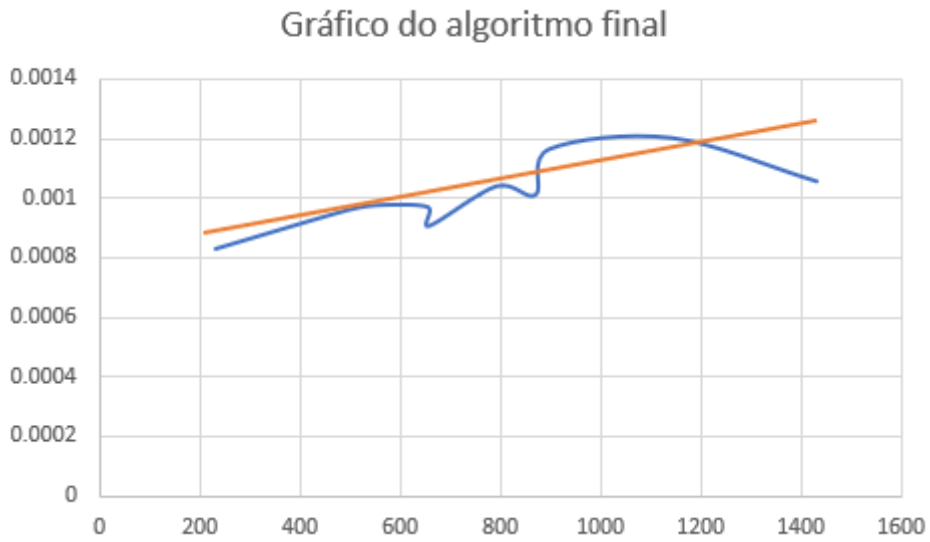
| | | | |
|---|---|---|---|
| caso01.txt | tempo = 0.00053199 | tamanho = 178833 | letra inicial = q |
| caso02.txt | tempo = 0.00055354 | tamanho = 399461207 | letra inicial = i |
| caso03.txt | tempo = 0.00061170 | tamanho = 15563117040 | letra inicial = m |
| caso04.txt | tempo = 0.00058017 | tamanho = 37554807516 | letra inicial = m |
| caso05.txt | tempo = 0.00057135 | tamanho = 26835107381 | letra inicial = a |
| caso06.txt | tempo = 0.00062603 | tamanho = 35172695271276 | letra inicial = w |
| caso07.txt | tempo = 0.00059491 | tamanho = 248961376085 | letra inicial = w |
| caso08.txt | tempo = 0.00073585 | tamanho = 48024951450717 | letra inicial = e |
| caso09.txt | tempo = 0.00077105 | tamanho = 7740687772924768 | letra inicial = v |
| caso10.txt | tempo = 0.00066770 | tamanho = 147634677711 | letra inicial = e |

# Conclusões

Dado o problema proposto e as diferentes soluções possíveis de serem encontradas, acreditamos que, embora ineficiente, a primeira implementação foi importante para o nosso entendimento sobre o custo de um algoritmo na prática, ao passo que não conseguimos analisar mais de dois casos de teste utilizando ela.

Utilizando os tempos de execução de nossa solução final, geramos um gráfico junto com a soma de todas as letras das sequências de cada caso de teste. O resultado confirma que a

complexidade do nosso algoritmo é linear, a qual pode ser representada em *Big O Notation* por O(n).




Enfim, ficamos satisfeitos com a rapidez da execução de nossa solução final, a qual é relativamente constante em tempo de execução e cumpre fielmente o que foi pedido.