Motivação

Estruturas de Dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Problema

Um sítio de conteúdo pretende fazer uma série de postagens com as palavras/expressões mais comuns em diversos idiomas, incluindo também listas especializadas para determinadas áreas (como engenharia, culinária, etc).

Para criar as listas foram levantados diversos corpus (coleções de textos) para cada possível postagem. Também foram contratados especialistas em cada idioma / área. Para ajudar os especialistas existe a necessidade de processar os textos e gerar uma lista preliminar das palavras/expressões mais frequentes.

Uma equipe já pré-processou os corpus e gerou arquivos de texto contendo uma palavra/expressão em minúsculo por linha. Cada arquivo tem entre 1.000 e 1.000.000 de palavras. A equipe também já criou uma função para ler o arquivo e devolver uma lista com cada palavra/expressão de um arquivo.

Agora é preciso um programa que analise uma lista e gere uma tabela com as palavras/expressões mais frequentes. A quantidade de itens na tabela pode variar, dependendo do corpus e da equipe de especialistas que vai usar o programa.

Determinar um número específico de palavras/expressões mais frequentes em uma lista de palavras/expressões.

Definição de tipos de dados

Cada palavra/expressão será representada por uma string.

A saída será uma lista de valores `PalavraVezes`.

```
@dataclass
class PalavraVezes:
    '''
    Representa uma palavra e quantidade de vezes
    que ela apareceu em um lista.
    '''
    palavra: str
    vezes: int
```

```
def encontra_mais_frequentes(palavras: list[str], m: int) -> list[PalavraVezes]:
    '''
    Encontra as *m* palavras mais frequentes em *palavras*.
    Se a frequência de duas palavras for a mesma, então a
    que vem primeiro em ordem alfabética aparece primeiro.
    Se *m* for maior que a quantidade de palavras distintas,
    então devolve a frequência de todas as palavras.
    Requer que m > 0.

    Exemplos
    >>> encontra_mais_frequentes(['casa', 'de', 'a', 'ti', 'de', \
                                  'a', 'casa', 'a', 'de'], 3)
    [PalavraVezes(palavra='a', vezes=3),
     PalavraVezes(palavra='de', vezes=3),
     PalavraVezes(palavra='casa', vezes=2)]
    '''
```

Implementação - Dinâmica de grupo

Agora precisamos fazer a implementação!

Forme equipes de até 5 pessoas.

Proponha um algoritmo para resolver o problema.

Faça o download do código inicial e escreva a implementação. Depois que os testes estiverem passando, execute o programa com o seguinte comando (que encontra as 10 palavras mais frequentes dentre as primeiras 1000 palavras do arquivo `palavras.txt`):

```
python mais-frequentes.py 10 1000 palavras.txt
```

Apresente o algoritmo para a turma.

Abordagem incremental direta:

- Temos uma lista de palavras e frequência (em ordem de maior frequência) e temos uma palavra. Precisamos atualizar a lista considerando a palavra:
  - Se a palavra está na lista, aumentamos a frequência em 1 e “movemos” a palavra para manter a lista em ordem.
  - Senão, adicionamos na lista a palavra com a frequência 1.

Implementação da abordagem incremental

```python
def encontra_mais_frequentes(palavras: list[str], m: int) -> list[PalavraVezes]:
    contagem: list[PalavraVezes] = []
    for p in palavras:
        # Procura o índice i de p na contagem
        i = 0
        while i < len(contagem) and contagem[i].palavra != p:
            i = i + 1
        # Encontrou o índice de p? Ou seja, p está na contagem?
        if i < len(contagem):
            contagem[i].vezes = contagem[i].vezes + 1
        else:
            contagem.append(PalavraVezes(p, 1))
        # Ajusta contagem para manter a ordem
        while i > 0 and contagem[i - 1] < contagem[i]:
            troca(contagem, i - 1, i)
            i = i - 1
    return contagem[:m]
```

Abordagem usando um plano (etapas):

- Calculamos a frequência de cada palavra
    - Podemos usar o método incremental: começamos sem nenhuma palavra e analisamos uma palavra por vez. Se a palavra já aparece, aumentamos sua frequência em 1, senão, adicionamos a palavra com frequência inicial de 1
- Ordenamos as palavras em ordem de maior frequência
    - Podemos usar a ordenação por seleção: encontramos a palavra com maior frequência e colocamos na posição 0 da lista, repetimos o processo para as posições 1, 2, ... até a última posição da lista.

Implementação da abordagem de etapas

```python
def encontra_mais_frequentes(palavras: list[str], m: int) -> list[PalavraVezes]:
    # Calcula a frequência de cada palavra
    contagem: list[PalavraVezes] = []
    for p in palavras:
        i = 0 # Procura o índice i de p na contagem
        while i < len(contagem) and contagem[i].palavra != p:
            i = i + 1
        # Encontrou o índice de p? Ou seja, p está na contagem?
        if i < len(contagem):
            contagem[i].vezes = contagem[i].vezes + 1
        else:
            contagem.append(PalavraVezes(p, 1))
    # Ordena por maior frequência
    for i in range(len(contagem)):
        m = i
        for j in range(i + 1, len(contagem)):
            if contagem[m] < contagem[j]:
                m = j
        troca(contagem, i, m)
    return contagem[:m]
```

Perguntas

Como avaliar qual desses algoritmos é mais adequado?

Se os algoritmos funcionam corretamente, porque algum seria mais adequado do que outro?

A implementação da abordagem incremental está disponível no arquivo `mais-frequentes-inc.py`.

Encontra as 20 palavras mais frequentes entre as primeiras 10.000 primeiras palavras do arquivo `palavras.txt`

```
$ python mais-frequentes-inc.py 20 10000 palavras.txt
Palavras mais frequentes:
the 620
of 340
...
Tempo de execução: 0.57 segundos
```

Parece bom!

Encontra as 20 palavras mais frequentes entre as primeiras 200.000 primeiras palavras do arquivo `palavras.txt`

```
$ python mais-frequentes-inc.py 20 200000 palavras.txt
Palavras mais frequentes:
the 13143
of 6228
...
Tempo de execução: 31.49 segundos
```

Demorou muito para executar...

A implementação da abordagem com etapas está disponível no arquivo `mais-frequentes-etapas.py`.

Encontra as 20 palavras mais frequentes entre as primeiras 10.000 primeiras palavras do arquivo `palavras.txt`

```
$ python mais-frequentes-etapas.py 20 10000 palavras.txt
Palavras mais frequentes:
the 620
of 340
....
Tempo de execução: 0.59 segundos
```

Parece bom!

Encontra as 20 palavras mais frequentes entre as primeiras 200.000 primeiras palavras do arquivo `palavras.txt`

```
$ python mais-frequentes-etapas.py 20 200000 palavras.txt
Palavras mais frequentes:
the 620
of 340
....
Tempo de execução: 28.32 segundos
```

Demorou muito para executar...

Nenhuma das duas abordagens é viável...

Podemos fazer melhor? Sim! Mas precisamos de estruturas de dados e métodos de ordenação eficientes, que é o que vamos fazer nessa disciplina!

Dentre as várias estruturas de dados e algoritmos que vamos estudar podemos destacar:

- Árvores binárias balanceadas de busca: mantêm uma coleção de itens ordenados que podem ser pesquisados e alterados de forma eficiente. Permitirá que o algoritmo incremental que nós vimos seja implementado de forma eficiente.
- Tabelas de dispersão: mantêm uma coleção de itens que podem ser pesquisados e alterados de forma eficiente.
- Algoritmo de ordenação por particionamento: ordena uma coleção de itens de forma eficiente. Junto com tabelas de dispersão, permitirá que o algoritmo por etapa que nós vimos seja implementado de forma eficiente.

Tá, mas quão eficiente podemos deixar os algoritmos que implementamos?

A biblioteca do Python disponibiliza um tipo chamado `dict` (dicionário), que é implementado com uma tabela de dispersão. Além disso, também temos a função `sort`, que é implementada com um algoritmo mais eficiente do que aquele que fizemos.

Podemos implementar nosso algoritmo por etapas com `dict` e `sort`.

Note que não vamos utilizar as coisas prontas do Python durante a disciplina, a ideia aqui é apenas mostrar quão eficiente o programa pode ser usando estruturas de dados e algoritmos de ordenação adequados.

Implementação eficiente

```python
def encontra_mais_frequentes(palavras: list[str], m: int) -> list[PalavraVezes]:
    # Calcula a frequência de cada palavra
    contagem: dict[str, int] = {}
    for p in palavras:
        if p in contagem:
            contagem[p] = contagem[p] + 1
        else:
            contagem[p] = 1

    # Cria uma lista de PalavrasVezes
    freqs = []
    for p, vezes in contagem.items():
        freqs.append(PalavraVezes(p, vezes))

    # Ordena a lista
    freqs.sort(reverse=True)

    return freqs[:m]
```

Essa implementação está disponível no arquivo `mais-frequentes-etapas-dict-sort.py`.

Encontra as 20 palavras mais frequentes entre as primeiras 200.000 primeiras palavras do arquivo `palavras.txt`

```
$ python mais-frequentes-etapas.py 20 200000 palavras.txt
Palavras mais frequentes:
the 13143
of 6228
...
Tempo de execução: 0.037 segundos
```

Conclusões

Mesmo que um algoritmo funcione corretamente e seja rápido para algumas entradas, ele pode ser inviável devido ao seu tempo de execução para entradas maiores.

Precisamos de uma forma para determinar o tempo de execução de um algoritmo sem precisar implementá-lo.

As estruturas de dados são essenciais para algoritmos eficientes.

Algoritmos de ordenação eficientes são importantes para a computação.