Aula 10: 11/SET/2018

Aulas passadas

Tudo começou com o seguinde problema apresentado na aula 08.

Problema: dada uma lista de palavras, contar quantas vezes cada palavra ocorre na lista.

O ingrediente fundamental para resolver esse problema foi a implementação de um dicionário através de uma classe `Dicio`. Os atributos de estado dessa classe foram duas listas: uma lista `chaves` e uma lista `valores`. Os métodos da `API` dessa classe foram `get(chave)` e `put(chave, valor)`. Ambos os métodos utilizaram o método `indice()` que era o *motor* da classe.

Na aula 08, na classe `Dicio` implementada em `dicio.py`, a lista `chaves` não era ordenada e as busca feita pelo método `indice()` eram sequenciais.

Na aula 09, a implementação da classe `Dicio` foi alterada. Na classe `Dicio` de `dicio_bin.py` passamos a manter a lista `chaves` ordenada no *motor* `indice()` substituimos busca sequencial por binária.

Foram feitas análises experimentais e analíticas do consumo de tempo dessas implementações e de uma implementação em `dicio_de_luxe.py` que utilizava o método `list.insert()` para descolar os elementos de `chaves` e `valores`.

Os resultados analíticos estão resumido na tabela a seguir.

O símbolo `~` é uma abreviatura de *proporcional a*. O fator de proporcionalidade está sendo omitido.

O pior caso corresponde a *busca com sucesso* (= encontramos a chave). O pior caso corresponde a *busca com fracasso* (= não encontramos a chave).

busca sequencial	melhor caso	pior caso
`get()`	`~1`	`~m`
`put()`	`~1`	`~m`
`indice()`	`~1`	`~m`
`criação`	`~n`	`~nm`

busca binária	melhor caso	pior caso
`get()`	`~1`	$\lg \mathtt{m}$
`put()`	`~1`	$\mathtt{m} + \lg \mathtt{m}$
`indice()`	`~1`	$\lg \mathtt{m}$
`criação`	`~n`	$\sim \mathtt{nm} + \mathtt{n} \lg \mathtt{m}$

A análise exeperimental mostrou que os melhoramentos surtiram efeito e que sempre que possível é preferível utilizarmos métodos ou funções nativas do Python.

Hoje

A classe nativa `dict` do Python e mais rudimentos de análise de algoritmos experimental e analítica.

Continuaremos as análise utilizando como laboratório o problema de ordenar os elementos de uma lista. Esse problema nos levará a muitas técnicas em análise e projeto de algoritmos que serão vistos nas próximas aulas.

Arquivos

No diretório da aula três implementações de dicionário (`Dicio`):

- `dicio.py`: busca sequencial
- `dicio_bin.py`: busca binário com implementação do deslocamento
- `dicio_de_luxe.py`: busca binária com delocameto sendo feito com `insert()`

O programa que usa todos é o `conta_Dicio.py`. Para usar algum desses dicionários devemos alterar a importação no início do arquivo.

```
from dicio import Dicio
```

Para experimetação temos o programa `experimentos.py`

```
% python experimentos.py
Uso: experimentos.py <arquivo> [-d<k> | -m<tam>]
      <arquivo> nome de um arquivo com uma lista de strings
     -d<ind> ind indica o dicionário a ser usado
        ind = 0: usa o dicionário com busca sequencial (default)
        ind = 1: usa o dicionário com busca binária
        ind = 2: usa o dicionário com busca binária e método insert()
        ind = 3: usa o dicionário nativo do Python
     -m<tam> testa com lista com 2^k <= <tam> elementos (default é no. de palavras no arquivo)
```

`main()`: conta palavras

Arquivo: `conta_Dicio.py`

Dado um arquivo com uma lista de palavras, conta o número de ocorrências de cada palavra na lista.

O trecho relevante é

```
dicio = Dicio()
for pal in lst_pals:
    valor = dicio.get(pal)
    if valor == None:
        dicio.put(pal, 1)
    else:
        dicio.put(pal, valor+1)
```

Classe `dict`

Análise experimental com dicionário nativos.

```
% python experimentos.py pal/hugo.pal -d3 -m300000
lendo as palavras no arquivo
leitura encerrada
criando listas de palavras
lista de palavras criada

  Análise experimental para criar dicionários
  -------------------------------------------

      m    tempo(s)  len(dicio)     tempo/m
  -----------------------------------------
    256     0.00        162        0.0000004
    512     0.00        279        0.0000003
   1024     0.00        476        0.0000003
   2048     0.00        833        0.0000003
   4096     0.00       1422        0.0000004
   8192     0.00       2312        0.0000004
  16384     0.01       3584        0.0000004
  32768     0.01       5444        0.0000003
  65536     0.02       8242        0.0000004
 131072     0.05      10947        0.0000004
 262144     0.10      16133        0.0000004
```

Parece que o tempo por operação independe do tamanho do dicionário!

De fato, o consumo de tempo **esperado** é constante.

O preço a pagar é que as chaves **não** tem ordem alguma

Dicionário em Python

Python possui dicionários como um tipo nativo.

Uma maneira de criar um dicionário é começar com o dicionário vazio e adicionar pares chave-valor. O dicionário vazio é denotado por `{}`

Métodos de dicionários

Dicionários possuem vários métodos nativos úteis. A seguinte tabela fornece um resumo e mais detalhes podem ser encontrados em Python Documentation.

Método	Parâmetros	Descrição
`keys`	nenhum	Retorna uma vista das chaves no dicionário
`values`	nenhum	Retorna uma vista dos valores no dicionário
`items`	nenhum	Retorna uma vista dos pares chave-valor no dicionário
`get`	key	Retorna o valor associado com a chave; ou None
`get`	key,alt	Retorna o valor associado com a chave; ou alt

`main()`: com classe `dict`

Arquivo: `conta_dict.py`

Dado um arquivo com uma lista de palavras, conta o número de ocorrências de cada palavra na lista.

O trecho relevante é

```
dicio = dict() # ou {}
for pal in lst_pals:
    valor = dicio.get(pal)
    if valor == None:
        dicio[pal] = 1
    else:
        dicio[pal] = valor + 1
```

Resto da função `main()`

```
palavra = input(PROMPT).strip()
while palavra != "":
    if palavra == MOSTRE:
        print(dicio)
    elif palavra == CHAVES:
        print(dicio.keys())
    elif palavra == VALORES:
        print(dicio.values())
    elif palavra == ITENS:
        print(dicio.items())
    elif palavra == LEN:
        print(len(dicio))
    elif palavra == MAX:
```

```
            chave, valor = maior_valor(dicio)
            print('%s: %s' %(chave,str(valor)))
        else:
            valor = dicio.get(palavra)
            print('%s: %s' %(palavra,str(valor)))
        palavra = input(PROMPT).strip()

#-----------------------------------------
def maior_valor(dicio):
    ''' (dicio) -> chave, valor

    Recebe um dicionário dicio em que as chaves
    são strings e os respectivos valores são
    comparáveis e retorna o par chave,valor que
    tem o maior valor
    '''

    chave = ''
    valor = 0
    for key in dicio:
        val = dicio[key]
        if val > valor:
            valor = val
            chave = key
    return chave, valor
```

Problema

Escreva um trecho de código que recebe uma lista `v[0:i]` tal que `v[0:i-1]` é crescente e rearranja seus elementos de maneira que `v[0:i]` seja crescente.

```
pivo = v[i]
j = i-1
while j >= 0 and v[j] > pivo:
    v[j+1] = v[j]
    j -= 1
v[j+1] = pivo
```

Problema

Usando o trecho de código anterior, escreve uma função que recebe uma lista e rearranja seus elementos de tal forma que a lista fique crescente. sejam o número de ocorrências de cada palavra no texto.

Ordenação de inserção

```
v     n=12
+----+----+----+----+----+----+----+----+----+----+----+----+----+
| 50 | 35 | 99 | 38 | 55 | 20 | 44 | 10 | 40 | 65 | 25 | 35 |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+
   0    1    2    3    4    5    6    7    8    9   10   11   12
```

Simular:

```
#---------------------------------------
def insercao(v):
1    n = len(v)
2    for i in range(1, n):
3        pivo = v[i]
         # encontre posição j tal que v[j] <= pivo < v[j+1]
4        j = i-1
5        while j >= 0 and v[j] > pivo:
6            j -= 1
         # insira o elemento na posição j+1
7        for k in range(i, j+1, -1):
8            v[k] = v[k-1]
         # coloque o pivo na sua posição
9        v[j+1] = pivo

def insercao(v):
    n = len(v)
    for i in range(1, n):
        pivo = v[i]
        j = i-1
        while j >= 0 and v[j] > pivo:
            v[j+1] = v[j]
```

```
            j -= 1
        v[j+1] = pivo
```

Invariantes da ordenação por inserção

- a lista é uma permutação da original
- na linha do `while i < n-1:` vale que `v[0:i-1]` é crescente

Consumo de tempo

```
linha    número de execuções da linha
  1         = 1
  2         = n
  3-4       = n-1
  5-6       <= 1 + 2 + ... + n-1 = (n*(n-1))/2
  7-8       <= (n*(n-1))/2
  9         = n-1
-----------------------------------------------
Total       = n^2 + 2n -1  = O(n^2)
```

O consumo de tempo da ordenação inserção no melhor caso é proporcional a n.

O consumo de tempo da ordenação por inserção no pior caso é proporcional a n^2.

O consumo de tempo da ordenação por inserção é $O(n^2)$.

Notação assintótica

Sejam $\mathtt{T(n)}$ e $\mathtt{f(n)}$ funções dos inteiros nos reais.

Dizemos que $\mathtt{T(n)}$ é $\mathtt{O(f(n))}$ se existem constantes positivas $\mathtt{c}$ e $\mathtt{n_0}$ tais que

$$\mathtt{T(n)} \leq \mathtt{c\,f(n)}$$

para todo $\mathtt{n} \geq \mathtt{n_0}$.

Mais informal

$\mathtt{T(n)}$ é $\mathtt{O(f(n))}$ se existe $\mathtt{c} > \mathtt{0}$ tal que $\mathtt{T(n)} \leq \mathtt{cf(n)}$, para todo *n suficientemente* GRANDE.

Ordenação por inserção binária

Acho que não devíamos falar mais nisso, pelo menos hoje. Precisamos de problemas e soluções simples para fazermos análise de algoritmos experimental e analítica.

```
def insercao(v):
1    n = len(v)
2    for i in range(1, n):
3        pivo = v[i]
         # encontre posição j tal que v[j] <= pivo < v[j+1]
4        j = i-1
5        while j >= 0 and v[j] > pivo:
6            j -= 1
         # insira o elemento na posição j+1
7        for k in range(i, j+1, -1):
8           v[k] = v[k-1]
         # coloque o pivo na sua posição
9        v[j+1] = pivo
```

```
def insercao(v):
1    n = len(v)
2    for i in range(1, n):
3        pivo = v[i]
         # encontre posição j tal que v[j] <= pivo < v[j+1]
5-6      j = busca_binaria(x, i, v)
         # insira o elemento na posição j+1
         for k in range(i, j+1, -1):
             v[k] = v[k-1]
         # coloque o pivo na sua posição
9        v[j+1] = pivo
```

```
def busca_binaria(x, n, v):
    '''(valor, list) -> int

    Recebe uma v crescente v[0:n] com n >= 1
    e um valor x e retorna um índice j em [0:n]
    tal que v[j] <= x < v[j+1]
    '''
    e, d, -1, n
    while e < d - 1:
        m = (e + d) // 2
        if v[m] <= x: e = m
        else: d = m
    return e
```

Consumo de tempo

```
linha    número de execuções da linha
```

```
  1         = 1
  2         = n
  3-4       = n-1
  5-6       <= lg 1 + lg 2 + ... + lg (n-1) <= n lg n
  7-8       <= (n*(n-1))/2
  9         = n-1
-----------------------------------------------
Total       = n^2/2 + n lg n + 7n/2 - 1  = O(n^2)
```

Consumo de tempo

O consumo de tempo da inserção binária no melhor caso é proporcional a $\mathtt{n} \lg \mathtt{n}$.

O consumo de tempo da inserção binária no pior caso é proporcional a $\mathtt{n}^2$.

O consumo de tempo da inserção binária é $\mathtt{O}(\mathtt{n}^2)$.

Apêndice

Análise experimental da aula passada

Implementação em `dicio.py`

```
% python experimentos.py pal/hugo.pal -d0 -m300000
lendo as palavras no arquivo
leitura encerrada
criando listas de palavras
lista de palavras criada

  Análise experimental para criar dicionários
  -------------------------------------------

      m    tempo(s)  len(dicio)     tempo/m
  -------------------------------------------
     256     0.00        162        0.0000143
     512     0.01        279        0.0000239
    1024     0.04        476        0.0000401
    2048     0.10        833        0.0000506
    4096     0.35       1422        0.0000843
    8192     1.05       2312        0.0001285
   16384     2.90       3584        0.0001769
   32768     7.41       5444        0.0002261
   65536    19.46       8242        0.0002969
  131072    44.32      10947        0.0003381
  262144   111.10      16133        0.0004238
```

Implementação em `dicio_bin.py`

```
% python experimentos.py pal/hugo.pal -d1 -m300000
lendo as palavras no arquivo
leitura encerrada
criando listas de palavras
lista de palavras criada

  Análise experimental para criar dicionários
  -------------------------------------------

      m    tempo(s)  len(dicio)     tempo/m
  -------------------------------------------
     256     0.00        162        0.0000102
     512     0.01        279        0.0000132
    1024     0.02        476        0.0000175
    2048     0.05        833        0.0000236
    4096     0.13       1422        0.0000312
    8192     0.33       2312        0.0000397
   16384     0.75       3584        0.0000456
   32768     1.71       5444        0.0000523
   65536     3.86       8242        0.0000590
```

```
  131072     7.20      10947        0.0000549
  262144    15.61      16133        0.0000595
```

Implementação em `dicio_de_luxe.py`

```
% python experimentos.py pal/hugo.pal -d2 -m300000
lendo as palavras no arquivo
leitura encerrada
criando listas de palavras
lista de palavras criada

  Análise experimental para criar dicionários
  ------------------------------------------

       m    tempo(s)  len(dicio)     tempo/m
  ------------------------------------------
     256     0.00        162        0.0000050
     512     0.00        279        0.0000056
    1024     0.01        476        0.0000060
    2048     0.01        833        0.0000066
    4096     0.03       1422        0.0000072
    8192     0.06       2312        0.0000075
   16384     0.13       3584        0.0000079
   32768     0.27       5444        0.0000082
   65536     0.58       8242        0.0000089
  131072     1.21      10947        0.0000092
  262144     2.44      16133        0.0000093
```

Métodos de `list`

```
Método   Parâmetros    Resultado   Descrição
-----------------------------------------
append   item          mutador     Acrescenta um novo item no final da lista
insert   posição,item  mutador     Insere um novo item na posição dada
pop      nenhum        híbrido     Remove e retorna o último item
pop      posição       híbrido     Remove e retorna o item da posição.
sort     nenhum        mutador     Ordena a lista
reverse  nenhum        mutador     Ordena a lista em ordem reversa
index    item          retorna idx Retorna a posição da primeira ocorrência do item
count    item          retorna ct  Retorna o número de ocorrências do item
remove   item          mutador     Remove a primeira ocorrência do item
```