20 Aula 20: 22/OUT/2019

20.1 Aulas passadas

Tratamos de projeto e análise de algoritmo tendo como pretexto o problema de ordenação. Vimos várias técnicas nesse contexto:

- algoritmos dinâmicos: ordenação por inserção
- divisão e conquista: `mergesort()` e `quicksort()`
- estruturas de dados: ordenação por seleção com max-heap se transforma no `heapsort()`

Expressão o consumo de tempo e espaço através de notação assintótica. O resultado, em termos de consumo de tempo foi

algoritmo	melhor caso	pior caso
inserção	$\mathtt{O(n)}$	$\mathtt{O(n^2)}$
inserção binária	$\mathtt{O(n \lg n)}$	$\mathtt{O(n^2)}$
seleção	$\mathtt{O(n^2)}$	$\mathtt{O(n^2)}$
mergesort	$\mathtt{O(n \lg n)}$	$\mathtt{O(n \lg n)}$
quicksort	$\mathtt{O(n \lg n)}$	$\mathtt{O(n^2)}$
heapsort	$\mathtt{O(n \lg n)}$	$\mathtt{O(n \lg n)}$

Também vimos análise exerimental para validar a análise assintótica.

20.2 Hoje

Consideraremos o seguinte problema

Problema: dada uma lista de número inteiros, determinar o número de trios que somam zero.

Por exemplo, para a lista `[30, -30, -20, -10, 40, 0, 10, 15]` temos que há 4 trios que somam zero:

```
 30  -30   0
 30  -20 -10
-30  -10  40
-10    0  10
```

Esse é o conhecido **3SUM problem** (Wikipedia) e, apesar de parecer artificial, está relacionado a várias tarefas computacionais fundamentais em geometria computacional. Segundo a Wikipedia

> *The current best known algorithm for 3SUM runs in* $O(n^2(\log\log n)^{O(1)}/\log^2 n)$ *time*

Hipótese simplificadora. A lista não possui dois números iguais.

Essa hipótese simplifica o código de algumas soluções.

O objetivo da aula de hoje é através desse problema discutirmos sobre o consumo de tempo de operações sobre os objetos das classes nativas `list` e `dict`.

Há tabelas com o consumo de tempo dessas classes de objetos no final dessas notas.

20.3 Programa e dados

Nessa aula utilizaremos o programa `cronometro.py` que mostra o consumo de tempo de 4 soluções:

- força bruta: consumo de tempo $O(n^3)$
- ordenação e busca binária: consumo de tempo $O(n^2 \lg n)$
- dicionários: consumo de tempo esperado $O(n^2)$
- quadrático: solução que faz uma variante de busca binária e tem consumo de tempo $O(n^2)$.

Os arquivos de dados para esse problema foram copiados das páginas do algs4 de Princeton e são: `8ints.txt`, `1Kints.txt`, `2Kints.txt`, `4Kints.txt`, `8Kints.txt`, `16Kints.txt`, `32Kints.txt`, `64Kints.txt`, `128Kints.txt`.

20.4 Força bruta

Solução que testa todas os possíveis trios de números.

Solução

```
def conta_fb(a):
    '''(list) -> int'''
    conta = 0
    n = len(a)
    for i in range(n):
        for j in range(i+1, n):
            soma = a[i] + a[j]
            for k in range(j+1, n):
                if soma + a[k] == 0: # (*)
                    conta += 1
    return conta
```

Consumo de tempo é dominado pelo número de execuções da linha `(*)`:

$$\begin{aligned}\sum_{i=0}^{n-3}\sum_{j=i+1}^{n-2}\sum_{k=j+1}^{n-1} 1 &= \sum_{i=0}^{n-3}\sum_{j=i+1}^{n-2}(n-j-1)\\ &= \sum_{i=0}^{n-3}(n-i-1)(n-i-2)/2\\ &\leq \sum_{i=0}^{n-3} n^2/2 = O(n^3)\end{aligned}$$

Análise experimental

Consumo de tempo é $O(n^3)$ e os resultados experimentais foram

```
     n       fb     cont   arquivo
  1000   20.06s        0    1Kints
  2000  162.31s        2    2Kints
```

```
4000 1394.72s      2    4Kints
```

1395,72 ~ 23 minutos.

Quanto tempo o programa gastaria para para resolver o problema com `128Kints`:

$T(128000) = 128000^3 = 32^3 \times 4000^3 = 32^3 \times T(4000) = 32768 \times 23$ minutos

Resposta: aproximadamente 523 dias

20.5 Busca binária

Esta solução se apoia em:

- `list.sort()`: consome tempo $O(n \lg n)$ e
- busca binária: consome tempo $O(\lg n)$

Para busca binária usaremos o modulo `Lib/bisect.py`: https://docs.python.org/3/library/bisect.html

```
    from bisec import bisect_left
    >>> from bisect import bisect_left
    >>> a = [1,2,3]
    >>> bisect_left(a,1)
    0
    >>> bisect_left(a,2)
    1
    >>> bisect_left(a,4)
    3
    >>> bisect_left(a,-1)
    0
    >>> bisect_left(a,3)
    2
```

Mais especificamente, usaremos uma adaptação de `index` da página de `bisect`

```
def index(a, x, lo, hi):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x, lo, hi)
    if i != len(a) and a[i] == x:
        return i
    returna None # raise ValueError
```

Atenção: Talvez não usar `lo` e `hi` torne a solução mais parecida com a solução com dicionário.

Solução

```
#-------------------------------------------------------
def conta_bb(a):
    '''(list) -> int'''
    conta = 0
    n = len(a)
    a.sort()   # O(n log n)
    for i in range(n):
        for j in range(i+1, n):
            soma = a[i] + a[j]
            k = index(a, -soma, j+1, n) # O(lg n)
            if k != None: conta += 1
    return conta
```

Consumo de tempo é $O(n \lg n) + O(n^2 \lg n) = O(n^2 \lg n)$.

Análise experimental

n	bb	cont
1000	0.30s	0
2000	1.21s	2
4000	4.99s	2
8000	20.33s	11
16000	85.39s	121

Quanto tempo o programa gastaria para para resolver o problema com `128Kints`:

$T(128000) = 128000^2 \lg 128000\ 8^2 16000^2 \lg 16000 = 64 \times T(16000) = 5440$ segundos

Resposta: aproximadamente 1 hora e meia

20.6 Dicionário

Utiliza um dicionário e tem consumo de tempo esperado $O(n^2)$.

Solução

```
#-------------------------------------------------------
def conta_dict(a):
    '''(list) -> int'''
    conta = 0
    n = len(a)
    d = {}
    for k in range(n): d[a[k]] = k # n x O(1) = O(n) esperado

    for i in range(n):
        for j in range(i+1, n):
            soma = a[i] + a[j]
            k = d.get(-soma) # O(1) esperado
            if k != None and k > j: conta += 1
    return conta
```

Consumo de tempo esperado é $\mathtt{O(n^2)}$.

Análise experimental

```
      n    dict     cont
   1000    0.11s       0
   2000    0.42s       2
   4000    1.80s       2
   8000    7.17s      11
  16000   29.18s     121
  32000  123.12s     990
  64000  511.08s    7627
 128000 2203.89s   61546
```

$2203s < 37$ minutos

20.7 Quadrática

Solução

```
#-------------------------------------------------------
def conta_n2(a):
    '''(list) -> int'''
    conta = 0
    n = len(a)
    a.sort()
    for i in range(n-2):
        x = a[i]
        e = i+1
        d = n-1
        while e < d:
            y = a[e]
            z = a[d]
            soma = x + y + z
            if soma == 0:
                conta += 1
                e += 1
                d -= 1
            elif soma > 0: d -= 1
            else: e += 1
    return conta
```

Consumo de tempo é $O(n^2)$.

Análise experimental

n	n2	cont
1000	0.08s	0
2000	0.32s	2
4000	1.32s	2
8000	5.22s	11
16000	21.92s	121
32000	88.58s	990
64000	355.86s	7627

20.8 Eficiência de operadores de `list`

Suponha que a lista `v` tem `n` itens e que `w` tem `k` itens. Tabela copiada de *Resolução de problemas em Python* (link). Ver também a página *TimeComplexity* (link)

Operação	Notação assintótica
`v[i]`	$O(1)$
`v[i] = x`	$O(1)$
`v.append(item)`	$O(1)$
`pop()`	$O(1)$
`pop(i)`	$O(n)$
`insert(i,item)`	$O(n)$
`del v[i]`	$O(n)$
`for item in v:`	$O(n)$
`item in v`	$O(n)$
`v[e:d]`	$O(k)$
`del v[e:d]`	$O(n)$
`v[e:d] = w`	$O(n+k)$
`v.reverse()`	$O(n)$
`v+w`	$O(k)$
`v.sort()`	$O(n \lg n)$
`v*k`	$O(nk)$

20.9 Eficiência de operadores de `dict`

Suponha que a lista `d` tem `n` itens `chave-valor`. Tabela copiada de *Resolução de problemas em Python* (link). Ver também a página *TimeComplexity* (link).

Operação	Notação assintótica (média)
`d.copy()`	$O(n)$
`d[chave]`	$O(1)$
`d[chave]=valor`	$O(1)$
`del d[chave]`	$O(1)$
`for chave in d:`	$O(n)$
`chave in d`	$O(1)$