23 Aula 23: 31/OUT/2019

23.1 Aulas anteriores

Na aula 21 vimos como resolver o problema da subsequência comum mais longa (*Longest Common Substring* = LCS) utilizando a técnica de **programação dinâmica** (= recursão com tabela).

Já na última aula conversamos sobre a classe nativa `set` e sobre a sua eficiência. `set` é uma espécie de `dict` em que o valor não importa, somente a chave importa. O consumo de tempo esperado de `set` e `dict` são constantes (=`O(1)`) para as operações básicas como `x in set` e `x in dict`. Na aula consideramos o problema dos anagramas em que havia um dicionário em que as chaves eram da classe `str` e os valores `set`.

23.2 Hoje

Continuaremos a desenvolver as habilidades de resolver problemas computacionais, agora explorando as ferramentas e algoritmos que já conhecemos, para resolver problemas cada vez mais complexos. Em particular, ao invés de ver *ordenação* como problema, veremos mais uma vez *ordenação* como uma ferramenta.

Utilização de `list.sort()` e `sorted()` do Python.

23.3 Arquivos

Consideraremos o problema LRS de uma dada string `s`. Para os testes `s` será uma string com 1 a 10 milhões de dígitos de π ou Para esse problema temos os seguintes programas no diretório `py`:

- `lrsForcaBruta.py`: resolve o LRS testando essencialmente todos os pares de sufixos de `s`;
- `lrs.py`: usa ordenação e testando apenas `n` pares de sufixos; trava quando `s` tem 130 mil dígitos;
- `lrsX.py`: idêntico ao programa anterior, usa uma classe `MyString` em que fatias são vistas e não clones. Utilizando `MyString` é possível encontrar a LRS dos 10 milhões de dígitos de π
- `mystring.py`: contém uma implementação simples da classe `MyString` com tudo que é necessário para ordenar uma lista de sufixos que são dessa classe;
- `doublingTestFB.py`: testador para a solução força bruta, o consumo de tempo é proibitivo;
- `doublingTest.py`: testador para a solução que usa ordenação e fatias de strings, o consumo de espaço é proibitivo;
- `doublingTestX.py`: utiliza a classe `MyString` e conseguimos encontrar a LRS dos primeiros 10 milhões de dígitos de π

Para testes pequenos é possível utilizar os arquivos `*.txt` no diretório `py`:

```
aaaa.txt  abcd.txt  acgt.txt  tiny.txt
```

Há mais arquivos para testes no diretório `txt`:

```
chromosome11-human.txt  mobydick.txt  moby.txt      pi-1024.txt
pi-10million.txt        pi-128.txt    pi-16384.txt  pi-1million.txt
pi-2048.txt             pi-256.txt    pi-32768.txt  pi-32.txt
pi-4096.txt             pi-512.txt    pi-64.txt     pi-65536.txt
pi-8192.txt
```

Cuidado: executar `lrs.py` com um arquivo de 130 mil caracteres travou meu computador.

23.4 Motivação

Aplicações de ordenação. O Google exibe os resultados da pesquisa em ordem decrescente de "importância", uma planilha exibe colunas ordenadas por um campo específico, o `Matlab` classifica os autovalores de uma matriz simétrica em ordem decrescente. Ordenação também surge como uma subrotina crítica em muitos aplicativos que parecem não ter nada a ver com ordenação, incluindo:

- compressão de dados,
- computação gráfica: fecho convexo, par mais próximo,
- biologia computacional: *longest repeated substring* (LRS), tópico da aula de hoje
- agendar tarefas para minimizar a soma ponderada dos tempos de conclusão,
- ...

Historicamente, ordenação era mais importante para aplicações comerciais, mas também desempenha um papel importante na infraestrutura de computação científica. A NASA e a comunidade de mecânicos de fluidos usam ordenação para estudar problemas no fluxo rarefeito; esses problemas de detecção de colisão são especialmente desafiadores, pois envolvem dez bilhões de partículas e só podem ser resolvidos em supercomputadores em paralelo. Técnicas de ordenação são usadas em alguns códigos eficientes para simulação de corpos em movimento. Outra aplicação científica importante da ordenação é o balanceamento de carga dos processadores de um supercomputador paralelo. Os cientistas utilizam algoritmos de ordenação inteligente para realizar o balanceamento de carga em tais sistemas.

Fonte: Sorting and Search

23.5 Ordenação nativa

Python inclui duas operações para ordenação. O método `sort()` do tipo de dados nativo `list` que reorganiza os itens da lista em ordem crescente. Esse método realiza ordenação *inplace* e é um método mutador (modifica a lista).

Por outro lado, o Python possui ainda um função nativa chamada `sorted()` que não modifica a lista recebida. A função retorna uma nova lista contendo os itens em ordem crescente.

O uso do método `sort()` e da função `sorted()` é ilustrado logo abaixo.

```
>>> a = [3, 1, 4, 1, 5]
>>> b = sorted(a)
>>> a
[3, 1, 4, 1, 5]
>>> b
[1, 1, 3, 4, 5]
>>> a.sort ()
>>> a
[1, 1, 3, 4, 5]
>>>
```

A ordenação do Python usa uma versão do mergesort. É provável que seja substancialmente mais rápida (10-20 $\times$) do que o `merge.py` que implementamos em aula porque usa uma implementação de baixo nível que não é composta no Python, evitando assim a sobrecarga substancial que o Python impõe a si próprio. Como em nossas implementações de ordenação, você pode usar o `sorted()` do Python com qualquer tipo de dados comparável, como os tipos de dados `str`, `int`, `float`, `set` e `dict` nativos do Python.

23.6 Problema auxiliar

Para aquecer os motores considere o seguinte problema:

Problema. Dadas strings `s` e `t` encontrar a mais longa substring que aparece no início de ambas.

Exemplo:

```
s = "a a c a a g t t t a c a a g c"
     * * * * * * * * * * * * * * *
t = "a a c a a g t t t a c a a g t t t a c a a g c t a g c"
```

Solução

```
def lcp(s, t):
    '''(str, str) -> str
    RECEBE duas strings s e t e RETORNA o prefixo comum mais longo
    de s e t
    '''
    n = min(len(s), len(t))
    for i in range(n):
        if s[i] != t[i]:
            return s[0:i]
    return s[0:n]
```

Discussão

O consumo de tempo de pior caso dessa função é `O(n)`. No caso de strings `s` e `t` que são substrings dos dígitos de π o prefixo comum é pequeno e o consumo de tempo é essencialmente constante, como mostrarão os experimentos.

```
python lrsX.py < ../txt/pi-10million.txt
main(): lendo a string ...
main(): string lida
main(): procurando uma LRS...
main(): LRS encontrada...
'18220874234996'
elapsed time     = 501.161
```

23.7 Problema

Longest Repeated Substring (LRS). Dada uma string `s`, encontrar a mais longa substring que aparece pelo menos duas vezes em `s`.

Exemplos

```
s = "a a c a a g t t t a c a a g c"
       * * * * *       * * * * *

lrs = "a c a a g c"

                      + + + + + + + + + + + + + +
s = "a a c a a g t t t a c a a g t t t a c a a g c t a g c"
       * * * * * * * * * * * * * *

lrs = "a c a a g t t t a c a a g"
```

Dígitos de π: comprimento do LRS nos primeiro 10 milhões de dígitos de π é 14: `'3186120489507'`.

Criptografia: verificação de repetições podem ajudar a quebrar o código.

23.8 Solução Força bruta

Fato: `lrs(s) = lcp(s[i:], s[j:])` para algum `i` e `j`.

Ideia: Aplique `lcp()` em todos pares de sufixos de `s`.

```
def lrsFB(s):
    '''(str) -> str
    Recebe uma string s e RETORNA uma substring repedida mais longa de s.
    '''
    n = len(s)
    lrs = ''
    for i in range(n-1):          # O(n)
        si = s[i:]                # O(n^2)
        for j in range(i+1,n):    # O(n^2)
            sj = s[j:]            # O(n^3)
            x = lcp(si, sj)       # O(n^3) para prefixos comuns longos
            if len(x) > len(lrs):
                lrs = x
    return lrs
```

23.8.1 Discussão

Para a análise do consumo de tempo seria mais simples contar o número de chamadas de `lcp()` sem esquecer do preço de clonar fatias.

A função faz $O(n^2)$ chamadas da função `lcp()`. Cada uma pode consumir, no pior caso, tempo $O(n)$. No caso dos dígitos de π, do ponto de vista prático, o consumo de tempo é constante.

O fatiamento faz o consumo de tempo se $O(n^3)$ independentemente do consumo de tempo de `lcp()`.

23.8.2 Experimentos

Pelos experimentos é possível verificar que o consumo de tempo é $O(n^2)$ e não $O(n^3)$. O que ocorreu? As fatias clonadas são reutilizadas?

```
% python doublingTestFB.py < ../txt/pi-1million.txt
main(): lendo a string ...
main(): string lida
      n      tempo  LRS
     32     0.000s  '26'
     64     0.002s  '592'
    128     0.012s  '592'
    256     0.033s  '5028'
    512     0.125s  '5028'
   1024     0.501s  '23846'
   2048     2.077s  '949129'
   4096     8.423s  '949129'
   8192    36.563s  '7111369'
  16384   177.626s  '52637962'
  32768   761.726   '84865383'
```

Usando uma entrada "pior caso", com 8 mil zeros, temos:

```
% python doublingTestFB.py < ../txt/zeros-2k.txt
main(): lendo a string ...
main(): string lida
      n      tempo  LRS
     32     0.001s  '00000000000000000000000000000000'
     64     0.005s  '0000000000000000000000000000000000000000000000000000000000000000'
    128     0.030s  '00000000000000 ... 00'
    256     0.217s  '000000000000000000 ... 00'
    512     1.602s  '0000000000000000000000 ... 00'
   1024    12.665s  '00000000000000000000000000 ... 00'
   2048   106.062s  '0000000000000000000000000000000000 ... 00'
```

23.9 Solução mais eficiente

Fato. suponha `sufixo[]` é a lista com todos os sufixos de `s`. Se `sufixo[i] <= sufixo[j]` para `i < j`, então, `lrs(s) = lpc(sufixo[i], sufixo[i+1])` para algum `i`.

```
 i       vetor de sufixos          sufixos ordenados        index[i]
--    ----------------------    ----------------------    --------
 0    A B R A C A D A B R A !    !                         11
 1    B R A C A D A B R A !      A !                       10
 2    R A C A D A B R A !        A B R A !                 7
 3    A C A D A B R A !          A B R A C A D A B R A !   0
 4    C A D A B R A !            A C A D A B R A !         3
 5    A D A B R A !              A D A B R A !             5
 6    D A B R A !                B R A !                   8
 7    A B R A !                  B R A C A D A B R A !     1
 8    B R A !                    C A D A B R A !           4
 9    R A !                      D A B R A !               6
10    A !                        R A !                     9
11    !                          R A C A D A B R A !       2
```

Ideia: ordenar os sufixos de `s` e aplicar `lcp()` apenas nos pares consecutivos na ordem.

```
    A B R A !                 7
    * * * *
    A B R A C A D A B R A !   0
```

```
def lrs(s):
    '''(str) -> str
    RECEBE uma string s e RETORNA uma substring repedida mais longa de s.
    '''
    n = len(s)

    # cria um listar com os sufixo de s
    sufixo = [s[i:] for i in range(n)] # O(n^2)

    # ordena os sufixo de s
    sufixo.sort()            # O(n lg n)

    # encontra o lrs comparando sufixo adjacentes
    lrs = ''
    for i in range(n-1): #O(n)
            x = lcp(sufixo[i], sufixo[i+1])  O(n^2)
            if len(x) > len(lrs):
                lrs = x

    return lrs
```

Discussão

A criação dos sufixos consome tempo $O(n^2)$ já que as fatias são clones. Assim, o consumo de tempo da função é $O(n^2)$.

O consumo de espaço é $O(n^2)$. Isso torna a função proibitiva para strings longas.

Experimentos

Os experimentos mostram um consumo de tempo quadrático

```
% python doblingTest.py < ../txt/pi-1million.txt
main(): lendo o string ...
main(): string lida.
      n      tempo  LRS
     32     0.000s  '26'
     64     0.001s  '592'
    128     0.002s  '230'
    256     0.002s  '0582'
    512     0.006s  '0348'
   1024     0.009s  '23846'
   2048     0.009s  '922796'
   4096     0.013s  '284886'
   8192     0.028s  '7111369'
  16384     0.074s  '52637962'
  32768     0.224s  '23533829'
  65536     0.774s  '201890888'
 131072     2.828s  '008173039'
 O COMPUTADOR TRAVOU AQUI
```

23.10 Solução mais eficiente ainda

A próxima solução é a assintoticamente mais eficiente. Essencialmente $O(n \lg n)$ para sequências sem substrings repetidas longas. Esse é o caso dos dígitos de π.

A função `lrsX()` é a mesma, apenas utilizamos uma classe `MyString` de fabricação própria em que fatias são **vistas** e não **clones**.

```
def lrsX(s):
    '''(MyString) -> MyString
    RECEBE uma string s e RETORNA uma substring repetida mais longa de s.
    '''
    n = len(s)

    # crie um lista com vista dos sufixo de s
    sufixo = [s[i:] for i in range(n)]   # O(n)

    # ordena os sufixo de s
    sufixo.sort()                        # O(n lg n)

    # encontra o lrs comparando sufixo adjacentes
    lrs = ''
    for i in range(n-1):
            x = lcp(sufixo[i], sufixo[i+1])
```

```
            if len(x) > len(lrs):
                lrs = x

    return lrs
```

Discussão

A implementação da classe `MyString` está mais adiante. Se fosse possível usar alguma biblioteca nativa do Python que fizesse o serviço teríamos uma solução mais eficiente na prática, apesar de assintóticamente o consumo de tempo ser o mesmo.

Experimentos

```
% python doublingTestX.py < ../txt/pi-10million.txt
main(): lendo a string ...
main(): string lida
      n      tempo  LRS
     32     0.000s  '26'
     64     0.001s  '592'
    128     0.002s  '230'
    256     0.004s  '0582'
    512     0.009s  '0348'
   1024     0.022s  '23846'
   2048     0.046s  '922796'
   4096     0.095s  '284886'
   8192     0.203s  '7111369'
  16384     0.439s  '52637962'
  32768     0.947s  '23533829'
  65536     2.079s  '201890888'
 131072     4.755s  '008173039'
 262144    10.226s  '4392366484'
 524288    20.545s  '1292345774'
1048576    46.855s  '756130190263'
2097152    99.422s  '756130190263'
4194304   211.528s  '082279930235'
8388608   452.009s  '3186120489507'
```

23.11 Apêndice

```
'''
Classe string que permite vista de uma substring.
Não está implementado do step em [start:stop:step]
'''
class MyString:
    def __init__(self, s, start=0, stop=None):
        self.s  = s
        self.start = start
        self.stop  = stop
        if stop == None: self.stop = len(s)

    def __str__(self):
        return self.s[self.start:self.stop]

    def __repr__(self):
        return self.s[self.start:self.stop]

    def __len__(self):
        return self.stop - self.start

    def __getitem__(self, key):
        if isinstance(key, int):
            return self.s[self.start+key]
        if isinstance(key, slice):
            if key.start is None: start = self.start
            else: start = self.start + key.start
            if key.stop  is None: stop  = self.stop
            else: stop = self.start + key.stop
            return MyString(self.s, start, stop)
        raise TypeError('Index must be int, not %s'%type(key))

    def __lt__(self, other):
        n = min(len(self), len(other))
        s       = self.s
        s_start = self.start
        o       = other.s
        o_start = other.start
        for i in range(n):
            if s[s_start+i] != o[o_start+i]:
                return s[s_start+i] < o[o_start+i]
        return len(self) < len(other)

    def __ge__(self, other):
        return not self < other

    def __gt__(self, other):
        n = min(len(self), len(other))
```

```
        s       = self.s
        s_start = self.start
        o       = other.s
        o_start = other.start
        for i in range(n):
            if s[s_start+i] != o[o_start+i]:
                return s[s_start+i] > o[o_start+i]
        return len(self) < len(other)

    def __le__(self, other):
        return not self > other


    def __eq__(self, other):
        if len(self) != len(other): return False
        n = len(self)
        s       = self.s
        s_start = self.start
        o       = other.s
        o_start = other.start
        for i in range(n):
            if s[s_start+i] != o[o_start+i]:
                return False
        return True

    def __ne__(self, other):
        return not self == other
```