

Python勉強会@HACHINONE

# 第9章

# 計算複雑性入門

# お知らせ

Python勉強会@HACHINOHEでは、ジョン・V・グッターグ『Python言語によるプログラミングイントロダクション』近代科学社、2014年をみんなで勉強しています。

この本は自分で読んで考えて調べると力が付くように書かれています。

自分で読んで考えて調べる前に、このスライドを見るのは、いわば**ネタバレ**を聞かされるようなものでもったいないです。

是非、本を読んでからご覧ください。

# 計算量

## Python勉強会@HACHINOHE

- 計算量
  - 入力データの量(値そのものでなく)に対し、計算量がどのように増大するか
    - 入力データの量(=長さ) $n$ は、入力データ $x$ のケタ数 $\log(x)$ に相当
  - 気にするのは時間がかかるときで、かからない場合はこの話は関係ない
    - 入力データ量がものすごく大きくなったとき、計算量がどんなふうに見えるか
    - 一番効く部分を考えればだいたいOK
  - ビッグオー O-記法:  $O(n)$  ... すごくでかいスケールで見たらどんな関数形に見えるか
- 時間計算量
  - どのくらい時間がかかるか
- 空間計算量
  - どのくらいメモリなどが必要か

# ケタ数

Python勉強会@HACHINOHE

- ケタ数は、最上位ケタの10の乗数(+1)

$$x = 827104773$$

$$= 8 \times 10^8 + 2 \times 10^7 + \dots + 3 \times 10^0$$

- そして10の乗数を求めるには、対数を使えばいい

$$x = 10^n$$

$$n = \log_{10} x$$



# ランダム・アクセス機械

Python勉強会@HACHINOHE

- 単純なコンピュータのモデル
- メモリに値が入っていて、アドレスを指定するとどのアドレスの値も同じスピードで操作できる
  - チューリングマシンはテープの上をヘッドが移動するので、データの格納位置によって速度が異なる
- 一度に一つずつ計算する
  - 並列計算しない
- ランダム・アクセス機械を想定して計算量を評価する

# 線形探索のステップ数

Python勉強会@HACHINOHE

- 線形探索: 前から順番に探す
- 入力Lとxによってステップ数が変わる
  - リストの前の方に含まれていた場合と、含まれていなかった場合

```
# -*- coding: utf-8 -*-
def linearSearch(L, x):
    """Lをリスト、xをオブジェクトとする
       Lにxが含まれていればTrue、そうでなければFalseを返す"""
    for e in L:
        if e == x:
            return True
    return False
```

# いろいろな計算時間

Python勉強会@HACHINOHE

- 最良計算時間: 一番早い場合
  - 線形探索では最初にマッチした場合で、 $O(1)$
- 最悪計算時間: 一番遅い場合
  - これに注目する人が多い
    - エドワード・A・マーフィーJr.米空軍大尉(当時)の法則的な
  - 線形探索では含まれていなかった場合で、 $O(n)$
- 平均計算時間: 平均値



# 文章の問題

Python勉強会@HACHINOHE

- 「 $f(x)$ の計算複雑さは $O(x^2)$ に属する」
  - 最悪計算時間の上界が $O(x^2)$
- 「 $f(x)$ の計算複雑さは $O(x^2)$ である」
  - 最悪計算時間の上界も下界も $O(x^2)$ : タイト・バウンド



# 計算量さまざま

Python勉強会@HACHINOHE

記法	名称	例
$O(1)$	定数計算時間	リストの長さ、乗算、偶奇判定
$O(\log n)$	対数計算時間	ソート済みの二分探索
$O(n)$	線形計算時間	ソートされていない場合の探索
$O(n \log n)$	対数線形計算時間	マージソート
$O(n^k)$	多項式計算時間	重み付き有向グラフの最短経路
$O(c^n)$	指数計算時間	巡回セールスマン

# 階乗を求める関数のステップ数

Python勉強会@HACHINOHE

- 階乗を求める関数: 約 $5n+2$ ステップ。この $n$ は引数

```
# -*- coding: utf-8 -*-
def fact(n):
    """nを自然数とする。n!を返す"""
    answer = 1           1回
    while n > 1:        1回
        answer *= n     2回
        n -= 1          2回
    return answer       1回
```

× 約 $n$ 回

- $O(n)$ ではなく $O(c^n)$ 
  - 変数の名前を変更し、引数 $x$ 、サイズを $n$ とする  
 $n=\log(x)$ 、逆に言うと $x=e^n$   
計算量  $5x+2 \sim e^n$

# ある計算の例

Python勉強会@HACHINOHE

- だいたい $1000 + x + 2x^2$  ステップ

```
# -*- coding: utf-8 -*-
def f(x):
    """xを自然数とする"""
    ans = 0

    for i in range(1000):
        ans += 1
    print 'ここまでの合計' + str(ans)

    for i in range(x):
        ans += 1
    print 'ここまでの合計' + str(ans)

    for i in range(x):
        for j in range(x):
            ans += 1
            ans += 1
    print 'ここまでの合計' + str(ans)
    print ans
```

】  $x$ によらず一定、1000回

】  $x$ 回

】  $2x^2$  回

$x$ が大きくなると、  
ほとんど $x^2$ ステップ  
と置いていい



# 非負の整数を文字列に変換

Python勉強会@HACHINOHE

- だいたい*i*のケタ数回、 $\log(i)$ ステップ
- データ量*n*は $\log(i)$ なので、 $O(n)$

```
# -*- coding: utf-8 -*-
def intToStr(i):
    """iを非負の整数とする。iを文字列に変換して返す"""
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i % 10] + result
        i = i // 10 # 小数点以下切り捨てる除算
    return result
```

**]** *i*が10で割れる回数  
=*i*の桁数 $\log(i)$

# 再帰で階乗を求める

Python勉強会@HACHINOHE

- 再帰的にx回呼び出される
- データ量 $n=\log(x)$ なので、 $x=e^n$ 、 $O(c^n)$

```
# -*- coding: utf-8 -*-  
def factorial(x):  
    """xを自然数とする。x!を返す"""  
    if x == 1:  
        return 1  
    else:  
        return x * factorial(x - 1) ] 再帰的にx回呼び出される
```


- 関数は呼び出されるたびにスタックフレームが必要
- 空間計算量はx個で、同様に $O(c^n)$

# 部分集合

Python勉強会@HACHINOHE

- L1の個数×L2の個数回で、 $O(\text{len}(L1)) \times O(\text{len}(L2))$

```
# -*- coding: utf-8 -*-
def isSubset(L1, L2):
    """L1、L2をリストとする。
    L1の各要素がL2に含まれればTrue、そうでなければFalseを返す"""
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```





# 共通部分

Python勉強会@HACHINOHE

- L1の個数×L2の個数回で、 $O(\text{len}(L1)) \times O(\text{len}(L2))$

```
# -*- coding: utf-8 -*-
def intersect(L1, L2):
    """L1、L2をリストとする。
       L1とL2の共通要素のリストを返す"""
    # 共通部分からなるリストを作る
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
                break
    # 重複した要素を取り除く
    result = []
    for e in tmp:
        if e not in result:
            result.append(e)
    return result
```

**L2の個数回** **L1の個数回**

**resultの個数回** **tmpの個数回**

# 2進数で表現した文字列を返す

Python勉強会@HACHINOHE


```
# -*- coding: utf-8 -*-
def getBinaryRep(n, numDigits):
    """nとnumDigitsを非負の整数とする。
       nをnumDigitsケタの2進数で表現した文字列を返す"""
    result = ''
    while n > 0:
        result = str(n % 2) + result
        n = n // 2
    if len(result) > numDigits:
        raise ValueError('ケタ数が不足')
    for i in range(numDigits - len(result)):
        result = '0' + result
    return result
```

# すべての部分集合

Python勉強会@HACHINOHE

- $O(2^{\text{len}(L)})$

```
# -*- coding: utf-8 -*-
def genPowerset(L):
    """Lをリストとする。すべての部分集合のリストを返す"""
    powerset = []
    for i in range(0, 2 ** len(L)):
        binStr = getBinary(i, len(L))
        subset = []
        for j in range(len(L)):
            if binStr[j] == '1':
                subset.append(L[j])
        powerset.append(subset)
    return powerset
```



2のLの個数乗回



# 考え方

## Python勉強会@HACHINOHE

- 要素数個の箱を用意し、0と1のあらゆる組み合わせ(0から $2^{\text{要素数}} - 1$ までのすべての値の2進数表現)を入れる
- 0が要素がなく、1があるとみなしてリストを作る

値	値の2進数表現		['a', 'b', ..., 'z', ...]だったら												
0	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	...	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	[]
0	0	0	0	0	0	0	0								
0	0	0													
	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	...	<table border="1"><tr><td>0</td><td>0</td><td>1</td></tr></table>	0	0	1	['z']
0	0	0	0	0	0	0	0								
0	0	1													
	⋮														
	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	1	0	0	0	0	0	...	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	['a', 'c']
1	0	1	0	0	0	0	0								
0	0	0													
	⋮														
$2^{\text{要素数}} - 1$	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	1	1	1	1	...	<table border="1"><tr><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	['a', 'b', ..., 'z', ...]
1	1	1	1	1	1	1	1								
1	1	1													