計算量の話

中西 渉 *

2022 年 12 月 28 日

1　はじめに

1.1　概要

コンピュータは人間と比べてはるかに高速の計算を行うが，だからといってどんな計算でもできるというものではない。非常に多くのデータや巨大な数値を扱うにはそれ相応の時間がかかるものだ。この講座では時間のかかる処理を実感・比較することによって，計算量の感覚を身につけることを目標とする。

1.2　準備

プログラミング環境は授業と同じように Spyder を使って Python で行うものとする。使用するプロジェクトは筆者のサイトに置いてある。Web ブラウザで `https://watayan.net` の「過去の発表資料」のページの「計算量の話」のところにある「プロジェクト」のリンクをクリックし，「プログラムで開く (unzip)」を選んで OK を押すとホームに展開される。そのあとは Spyder のメニューの「プロジェクト」→「プロジェクトを開く」で，自分のホームにある「計算量の話」のディレクトリを選択すればこのプロジェクトが開かれる。`example??.py` は以下の例??のプログラムである。

2　計算量

2.1　時間計算量・空間計算量

コンピュータである処理が実行できるかを考えるときに，考えるのは次の事柄である。

- どれだけの時間がかかるか
- どれだけのメモリを使うか

前者を**時間計算量**，後者を**空間計算量**というが，ここでは時間計算量についてだけ考えることにする（以下，単に**計算量**というときには時間計算量を指す)。

2.2　実行時間の計測

時間計算量を考えるためには，プログラムの実行時間を計測しなくてはいけない。しかし手作業でやりたくはないので `time` モジュールを使うことにする。`time.perf_counter()` という関数を処理の前と後で呼び出して，その差を計算することで実行時間を計測することにしよう。たとえば例 1 のようにすればいい。

例 1

```
import time

t1 = time.perf_counter()
(何らかの処理)
t2 = time.perf_counter()
print(t2 - t1)
```

* 名古屋高等学校

しかし，このあとは「何らかの処理」に注目したいので，それがプログラムの途中にあるのはわかりにくい。そこで例 2 のようなプログラムにして，N が入力されたあと f(N) の実行にかかる時間を計測することにする。例 2 では f(N) の中身は None だけであるが，これを計算量を考えたいプログラムに書き換えればいい[1]。なお，最後の行にある表示がごちゃごちゃしているのは，桁数を統一して見やすくするためである（10 桁のうち小数部分が 6 桁という指定）。

例 2

```
import time

def f(N):
    None

N = int(input())
t1 = time.perf_counter()
f(N)
t2 = time.perf_counter()
print("{:10.6f}".format(t2 - t1))
```

2.3 オーダー記法

入力する N の値によって実行時間はどのように変化するだろうか。例 2〜例 5 のプログラムについて，実際に実行して 100,1000,...,100000 を入力し，実行時間を後の表 1 に記録しなさい。ただし例 5 は 1000 までにとどめておくのが良い[2]。

なお，count += 1 は count = count + 1 と同じ意味で，変数名を 2 回書かなくてもいいのでよく使われる記法である。

例 3

```
def f(N):
    count = 0
    for i in range(N):
        count += 1
```

例 4

```
def f(N):
    count = 0
    for i in range(N):
        for j in range(N):
            count += 1
```

例 5

```
def f(N):
    count = 0
    for i in range(N):
        for j in range(N):
            for k in range(N):
                count += 1
```

表 1　実行時間の比較

N	例 2	例 3	例 4	例 5
100				
1000				
10000				
100000				

表 1 の実行時間を比較しよう。表 3 は N が 1 桁増えると実行時間も 1 桁増えるくらいではないだろうか。これは次のように考えることができる：

> N 回の繰り返しだから，i が増えるのが N 回，count が増えるのが N 回で変数の値が増えるのが $2N$ 回行われる。実行時間はそのような処理の回数におよそ比例するだろう。

このようなとき，実行時間は N にだいたい比例するということで $O(N)$ と表される（係数は比例定数に含まれるので書かない）。同様に表 4 は $O(N^2)$，表 5 は $O(N^3)$ となる。例 2 については，実行時間が N の影響を受けないということで $O(1)$ と表される。

実用的なプログラムを作るときには，想定される N の上限と，それに応じた計算量の見積もりが必要である。

1） None は何もしないという命令。
2） 筆者の PC では 10000 で数時間，100000 で数百日かかる見込み。

3 具体例

3.1 素数判定

教科書には素数判定のプログラムが掲載されていた。あれはそれなりにややこしいアルゴリズムになっているので，もっとも素朴なものから始めてだんだん近いものにしていく。

一番素朴な考え方は例 6 のように 2 から $N-1$ までで割ってみるというものだろう。この関数は N が素数ならば 1，そうでなければ 0 を返す関数とし，返す値を `flag` としている。これは繰り返し回数が $N-2$ であるから，計算量は $O(N)$ である。

例 6

```
def f(N):
    if N == 1:
        return 0
    flag = 1
    for i in range(2, N):
        if N % i == 0:
            flag = 0
    return flag
```

次に考えるのは，$N-1$ まで割らなくても $\frac{N}{2}$ まででで十分だというものだろう。このプログラムは例 7 のようになり，例 6 のおよそ半分の実行時間になるが，計算量は $O(N)$ のままだ。

例 7

```
def f(N):
    if N == 1:
        return 0
    flag = 1
    for i in range(2, N // 2 + 1):
        if N % i == 0:
            flag = 0
    return flag
```

実は $\frac{N}{2}$ まで割る必要はなくて，$\sqrt{N}$ で十分だったりする。そうするとプログラムは例 8 のようになり，計算量は $O(\sqrt{N})$ となる。

ちなみに教科書ではこれに加えて

- 2 で割り切れれば素数でない
- 2 で割り切れなければ 3〜$\sqrt{N}$ までの奇数だけで割ればいい

ということで繰り返しの回数はおよそ半分になるのだが，計算量は $O(\sqrt{N})$ のままである。

例 8

```
import math

def f(N):
    if N == 1:
        return 0
    s = int(math.sqrt(N))
    flag = 1
    for i in range(2, s + 1):
        if N % i == 0:
            flag = 0
    return flag
```

例 6 →例 7 →例 8 とプログラムを改良してきたわけだが，計算量から考えると例 6 →例 7 はどちらも $O(N)$ だから大したことはなくて，例 7 →例 8 は $O(\sqrt{N})$ になったから大したものだということになる。例 6 →例 7 も実行時間はおよそ半分にはなっているわけだが，N が 100 倍になればどちらも 100 倍，10000 倍になればどちらも 10000 倍である。一方例 8 は N が 100 倍になれば実行時間は 10 倍，10000 倍でも 100 倍なのだから「桁違い」の差がある。

余談：例 8 は `math.sqrt` を用いたが，次のように `while` を使ったループにすればその必要はなくなる。計算量はやはり $O(\sqrt{N})$ である。

例 9

```
def f(N):
    if N == 1:
        return 0
    flag = 1
    i = 2
    while i * i <= N:
        if N % i == 0:
            flag = 0
        i += 1
    return flag
```

3.2 数当てゲーム

1 から N までの乱数を発生させ，少ない質問回数でそれを当てるというゲームを考える。ただし質問は Yes か No で答えられるものだけだ。作戦として次の 3 つを考えたが，効率がいいのはどれだろう。

(1) 当てずっぽうに数を言って，偶然当たったら終了

(2) 1 から小さい順に当たりかどうかを尋ねて，当たったら終了

(3) まず $l=1$，$r=N$ とし，次の操作を繰り返す

  a) $l=r$ ならその値が当たりなので終了

  b) $b=\lfloor(l+r)/2\rfloor$ とする[3)]

  c) b が当たり以上なら $r=b$，そうでないなら $l=b+1$ として a) に戻る。

これらの実行時間を 1 回だけ計算するだけでは偶然によるものが大きいので，100 回繰り返すことにする（例 10〜例 12 は例 2 と違って，`f(N)` を 100 回繰り返すようになっている）。また，乱数を使うために `random` モジュールを `import` しているのだが，プリントでは省略する。使われている `random.randint(1, N)` は 1 以上 `N` 以下の整数の乱数を返す関数である[4)]。

(1) の「当てずっぽう」だとこうなる。

例 10

```
def f(N):
    a = random.randint(1, N)
    while a != random.randint(1, N):
        None
```

1 から順に尋ねる (2) は**線形探索**といい，プログラムは例 11 のようになるだろう。

3) $\lfloor x\rfloor$ は x の小数部分切り捨て。

4) `range(1, N)` と違い，`N` を含むことに注意。

例 11

```
def f(N):
    a = random.randint(1, N)
    b = 1
    while a != b:
        b += 1
```

(3) は**二分探索**といい，例 12 のように少しややこしくなる。

例 12

```
def f(N):
    a = random.randint(1, N)
    l = 1
    r = N
    while l != r:
        b = (l + r) // 2
        if b >= a:
            r = b
        else:
            l = b + 1
```

例 10〜例 12 についても実行時間を比較して表 2 にまとめてみよう。

表 2　実行時間の比較

N	例 10	例 11	例 12
100			
1000			
10000			
100000			

例 10，例 11 はともに $O(N)$ であるが，例 11 の方が若干効率がいい（運次第では例 10 が勝ることもあるかもしれないが）。一方，例 12 はかなり効率がいいらしく，例 10 や例 11 との差は N が大きくなればなるほど大きくなる。このしくみについて少し考察してみよう。

線形探索では 1 回で見つかることもあれば N 回で見つかる最悪のケースもある。平均すればおよそ $\frac{N}{2}$ 回なので，計算量は $O(N)$ になる。

二分探索における繰り返しの所要回数を x とすると，およそ $2^x=N$ となる。というのはこういうことだ。

- $N = 2$ のときは 1 回尋ねればわかるから 1 回。
- $N = 4$ のときは 1 回尋ねれば (1) になるから 2 回。
- $N = 8$ のときは 1 回尋ねれば (2) になるから 3 回。

…

数学 II で習うことだが，$2^x = N$ となる x を $\log_2 N$ と表す。つまり例 12 は $O(\log_2 N)$ ということで，格段に効率がいい[5]。素数判定で出てきた $O(\sqrt{N})$ は $O(N)$ と比べて桁数がおよそ半分であるが，$O(\log_2 N)$ はほぼ N の桁数に比例する。N の桁数を 10000 倍にすると $\sqrt{N}$ の桁数はおよそ 5000 倍になるが，$\log_2 N$ の桁数は 4 つ増えるだけである。

3.3 フィボナッチ数列

数列は数学でまだ学習していないので，ここでは次のような関数を考えることにする。

- $f(0) = 1$
- $f(1) = 1$
- $n \geqq 2$ のとき $f(n) = f(n-1) + f(n-2)$

3 つめの条件によって

$$\begin{aligned} f(2) &= f(1) + f(0) = 1 + 1 = 2 \\ f(3) &= f(2) + f(1) = 2 + 1 = 3 \\ f(4) &= f(3) + f(2) = 3 + 2 = 5 \\ f(5) &= f(4) + f(3) = 5 + 3 = 8 \\ &\cdots \end{aligned}$$

というようにどんどん先まで計算していくことができる。

これに対応する f(N) は上に書いたことをそのままプログラムにして例 13 のように書くことができる。このように関数の内部で自分自身を呼び出すことを**再帰**という。

5） $\log_2$ の 2 を「底」というが，底が変わっても log の値は定数倍しか違わないので，普通は底を省略して $O(\log N)$ という。

例 13

```
def f(N):
    if N < 2:
        return 1
    else:
        return f(N - 1) + f(N - 2)
```

これも実行時間を計測してみよう（プログラムは $f(N)$ の値も表示するようになっている）。ただ，表 3 は N が 900 まで書いてあるが，例 13 は 40 まででやめておいたほうがいい。

表 3 実行時間の比較

N	例 13	例 14	例 15
10			
20			
30			
40			
50			
100			
500			
900			

たかだか $N = 40$ でどうしてこんなに時間がかかってしまうのだろう。そこで例として，$f(5) = 8$ を計算する手間を考えてみる。

$$\begin{aligned} f(5) &= f(4) + f(3) \\ &= f(3) + f(2) + f(2) + f(1) \\ &= f(2) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1) \\ &= f(1) + f(0) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1) \end{aligned}$$

数学なら $f(2) + f(2) = 2f(2)$ として計算量を減らすのだが，プログラムでは $f(4)$ を計算しているときの $f(2)$ と $f(3)$ を計算しているときの $f(2)$ は別にやっているので，それぞれの分の手数がかかってしまう。その結果，$f(5)$ の値

と同じ個数の 1（つまり $f(1)$ と $f(0)$）にまで分解されてしまう（図 1 参照）。つまり $f(N)$ を計算するには $f(N)$ の値と同じくらいの回数の処理が必要になる。正確には

$$f(N) = \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{N+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{N+1}\right\}$$

なので，$f(N)$ の計算量は $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^N\right)$ である[6)]。そのため N が 10 増えると実行時間はおよそ 123 倍になる。例 13 を $N = 40$ まででやめておいた方がいいといったのはそういうことだ。

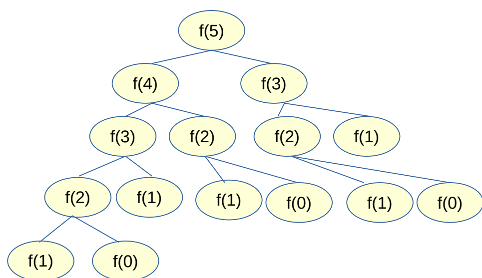

図 1　$f(5)$ の計算

計算量を減らす工夫を 2 つ紹介する。一つは最初に行ったように，$f(2)$，$f(3)$，$f(4)$，…というように順に計算していくというものだ。これはループを使えばいい。

例 14

```
def f(N):
    if N < 2:
        return 1
    else:
        a = 1
        b = 1
        for i in range(N - 1):
            c = a + b
            a = b
            b = c
        return c
```

もう一つは，すでに計算した値は覚えておいて，2 回以上計算しないというものだ。そのために Python の「辞書」という機能を使う。これは

`{キー 1:値 1, キー 2:値 2, キー 3:値 3,...}`

という構文で表されるデータで，キーに対応する値は「**辞書名 [キー]**」の形式で参照したり代入したりすることができる。例 15 では関数 `f` の中に `f.dict` という辞書を作っておいて，そこに既に計算した $f(N)$ の値を入れておくことにした。`f(N)` は，N が `f.dict` のキーにあれば[7)]計算せずに `f.dict[N]` の値を返し，キーになければ $f(N-1)+f(N-2)$ で計算する（この値は `f.dict` にも保存する）というプログラムになっている。

例 15

```
def f(N):
    if N in f.dict:
        return f.dict[N]
    a = f(N - 1) + f(N - 2)
    f.dict[N] = a
    return a
f.dict = {0:1, 1:1}
```

例 14 と例 15 はともに $O(N)$ であるから，例 13 にくらべてはるかに高速である。例 13 は N の指数関数であり，N が増えればかかる時間が急速に増えてしまう。

一般的に $O(N)$ や $O(N^2)$ のようなものは多項式時間，$O(2^N)$ のようなものは指数関数時間といわれる。$\sqrt{N}$ や $\log N$ は多項式ではないが，増え方が N よりも緩やかなので $O(\sqrt{N})$ や $O(\log N)$，$O(N \log N)$ なども多項式時間に含まれる。

N がある程度大きいときには指数関数時間のアルゴリズムが使いものにならないことがよくある（逆に N が小さいときに限っては有効なこともある）。多項式時間は指数関数時間に比べれば小さいことが多いが，さらにいえば次数が小さいに越したことはない。

6）およそ $O(1.62^N)$。

7）`N in f.dict` はこれを調べている。

4　おわりに

4.1　ソート

ソートの計算量は（比較ソートに限れば）大抵のものは $O(N\log N)$〜$O(N^2)$ である[8)]。たとえば簡単に実装できるバブルソートや挿入ソート，選択ソートは $O(N^2)$ であるから，速度が要求されるときはヒープソートやクイックソートなどが用いられる。しかしクイックソートは平均すれば $O(N\log N)$ なのだが，最初の並び順が最悪の場合には $O(N^2)$ になってしまうので注意が必要である。

4.2　素数判定

素数判定については $O(N)$ や $O(\sqrt{N})$ であるものを扱ったが，普通このような問題では N でなく $\log N$（大雑把にいえば N の桁数）の式でどう表されるかを考える。$N = 2^{\log_2 N}$ あるいは $\sqrt{N} = \sqrt{2}^{\log_2 N}$ なので，$O(N)$ や $O(\sqrt{N})$ はその意味では指数関数時間がかかる低速なアルゴリズムなのである。

2002 年にインド工科大学で，$\log N$ の多項式時間で素数判定ができることが証明された。その論文 “PRIMES is in P” が公開されたときには大きい話題となったことを覚えている。しかし，これで素数判定が速くなったというわけではない（多項式の次数が高すぎるため）。

4.3　暗号の話

計算量は小さければ小さいほどいい…と考えがちであるが，場合によっては計算量が大きくなくてはいけないものもある。例えば RSA 暗号方式では 2 つの大きい素数 p，q の積 $n = pq$ を公開鍵の一部として使う。復号に用いられる秘密鍵は p，q と公開鍵で作られるのだが，公開されている n を素因数分解して p と q を求めることができれば，それによって秘密鍵を生成できるので盗聴者が復号できてしまう。しかし巨大な数の素因数分解を（$\log n$ の）多項式時間で行うアルゴリズムは知られていないので，p や q を求めることは現実的な時間ではできず，RSA 暗号方式は安全であると考えられている。もっとも，量子コンピュータが実用化されれば可能だという考えもあるのだが。

4.4　競技プログラミング

AtCoder や情報オリンピックのような競技プログラミングでは，計算量の見積もりが重要である。たとえば次のような問題を考える[9)]。

> 非負正数列 $A = (a_1, a_2, \ldots, a_N)$ が与えられます。A の（添字が異なる）K 個の項の和として考えられる非負整数の集合を S とします。S に含まれる D の倍数の最大値を求めてください。ただし S に D の倍数が含まれない場合，代わりに -1 と出力してください。制約条件は以下の通りです。
>
> - $1 \leqq K \leqq N \leqq 100$
> - $1 \leqq D \leqq 100$
> - $0 \leqq a_i \leqq 10^9$

これを解くのに K 個の項の組み合わせすべてを調べようとすると，${}_NC_K$ 通りを調べることになる。N を固定したときの ${}_NC_K$ の最大値 ${}_NC_{\lfloor \frac{N}{2} \rfloor}$ は N の指数関数に近いので，制限時間内には終了しない。どういうアルゴリズムでそれを回避するかが競技プログラミングの醍醐味だと考えている。

8)　比較ソートは $O(N\log N)$ が下限。

9)　AtCoder Beginner Contest 281 D 問題。