計算量という考え方

名古屋高等学校　高１冬期進学講座
担当：数学科・情報科　中西渉

第１章　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量の計算でもできるというわけではない。巨大なデータや数値の計算には，やはりそれなりの時間がかかってしまう。少しでも高速に処理を済ませたいために，ハードウェアもソフトウェアも進歩を続けてきた。

しかし，何でも計算が速くできればいいというものではない。たとえば現代の暗号は時間をかければ解けるのだが，そのために途方もない時間が必要になるので暗号としての用が足せている。この場合は逆に計算量が小さくならない保証を求めていることになる。

この講座では，実際に時間のかかる処理を体験・比較することで，計算量を実感することを目標とする。

1.2　準備

Webブラウザで筆者の個人サイトである`https://watayan.net`の「過去の発表資料」のページの一番上にある「計算量という考え方」の「プログラム一覧のテキスト」を開きなさい。ここに使うプログラムのコードが書いてあるので，打ち込むのが面倒ならここからコピーしなさい。

次にSpyderを起動して，新規ファイルを一つ作りなさい。そこにこれからプログラムを書いていく（あるいはコピーする）ことにする。この講座ではこの１つのファイルだけを使い，中身を随時書き換えていく。

第2章　計算量

2.1　時間計算量・空間計算量

計算量というと主に次の2つを考える。

- 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。たとえば40人のクラスでばらばらに集めたプリントを，出席番号順に並べるときに次の2つの手順を考えてみる。

1. プリントの束を上から順番に見ていって出席番号1のプリントを抜き出して，裏向きにして隣に置く。次に同じように出席番号2のプリントを抜き出し，出席番号3のプリントを抜き出し…これを最後まで繰り返す。
2. 大きい机にプリント40枚分のスペースを用意して番号をつける。プリントの束の上から順に番号のところに置いていき，終わったら端から順に集める。

1.の方法は場所を取らないが時間がかかり，逆に2.の方法は短時間で済むが広い場所が必要になる。この例では，使う場所の広さが空間計算量，時間が時間計算量に相当する。

手順	時間計算量	空間計算量
1.	大	小
2.	小	大

以下，この講座では時間計算量だけを考えるので，これを単に **計算量** ということにする。

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。といっても手作業では不正確なので，ここでは `time` モジュールの `perf_counter` 関数を用いることにする。これを処理の前と後で呼び出してそ

の差を計算することで，実行時間がわかる。具体的には次のようになる。

```
import time

t1 = time.perf_counter()
（実行時間を計測する処理）
t2 = time.perf_counter()
print(t2 - t1)
```

しかし「実行時間を計測する処理」を今後何度も書き換えることを考えると，それがプログラムの途中にあるのはわかりにくい。そこで次のプログラムで，f(N)の中身だけを書き換えることにしよう（最後の表示がごちゃごちゃしているのは，表示の桁数を揃えるためである）。

このプログラムを入力またはコピーしなさい

```
import time
import random

def f(N):
    return N

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

これを入力（またはコピー）して実行してみなさい。実行すると入力待ちになるので，右下の「コンソール」をクリックして適当な整数を入力しなさい。入力した数と，実行時間が表示されるはずだ。

今後はこれのf(N)の部分だけを書き換えて実行し，入力したNの値と実行時間の関係について考えていく。

2.3 オーダー記法

入力するNの値によって実行時間がどのように変わるかを考えてみる。まずf(N)を書き換えないままで実行し，100，1000，10000，100000を入力したときの実行時間を次の表に記入しなさい（といっても，Nは実行時間に関係しないので，ほとんど同じ数値のはず）。

次にf(N)のところだけを以下の例1〜例3に置き換えて（手作業で書いてもいいし，コピーしてもいい），同様に100，1000，10000，100000を入力したときの実行時間をまとめなさい。ただし例3については1000までにとどめておくこと（筆者のPCでは10000で数時間，100000で数百日かかる見込み）。例2の100000も数分かかるだろう。

例1

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

例2

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

例3

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

N	元のまま	例1	例2	例3
100				
1000				
10000				×
100000				×

元のままのプログラムは`f(N)`の中で何もしていないから，`N`が増えても実行時間はほぼ変わらないはずだ。一方，例1や例2，例3は`N`が10倍になるごとに，実行時間は何倍になっているだろうか。理論上は例1は約10倍，例2は約100倍，例3は約1000倍になる。つまり例1の実行時間はおよそ`N`に比例し，例2の実行時間はおよそ`N**2`に，例3はおよそ`N**3`にそれぞれ比例する。実際に計算が行われている回数を大雑把に見積もると次のようになる。

- 例1は`i`も`count`もN回ずつ増え，計算が約$2N$回行われる。
- 例2は`i`がN回増え，そのたびに`j`も`count`もN回ずつ増え，計算が$N(1+2N) = N + 2N^2$回行われる。
- 例3は`i`がN回増え，そのたびに`j`もN回ずつ，そのたびに`k`も`count`もN回ずつ増え，計算が$N(1+N(1+2N)) = N + N^2 + 2N^3$回行われる。

そこで例1，例2，例3の計算量をそれぞれ$O(N), O(N^2)$，$O(N^3)$と表す（係数や最高次でない項は無視する）。このような表現を**オーダー記法**という。ちなみに元のままのプログラムは$O(1)$である。

計算量を表す式は多項式ばかりではない。たとえば整数の足し算や引き算の計算量はおよそ桁数に比例するので，計算量は$O(\log N)$である。なお，logは数学IIの「指数関数と対数関数」の単元で出てくるものなので，ここでは簡単に説明する。$a^x = b$のとき$x = \log_a b$と定義するものであるが，xが変化するとbは大きく変化するということを逆に考えると，bの増え方に比べてxの増え方は非常に緩やかである。つまり$\log N$の増え方は非常に緩やかである，ということだけがわかれば現時点では十分である。他にも$O(2^N)$とか$O(N!)$の計算量が必要なアルゴリズムもあって，このようなものは小さい数でしか計算ができない。

第3章　具体例

3.1　素数判定・素因数分解

ある整数が素数か素数でないかを判断するアルゴリズムを考える。たとえばNが素数かどうかを判定するには「2からN-1までの整数で割ってみて，どれかで割り切れれば素数でない，最後まで割り切れなかったら素数」と考えると次のようになる（素数なら1，素数でなければ0を返す関数にした）。

素数：例1

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

これの計算量は$O(N)$である。次のような改良をするとどうだろうか。

- 調べるのはN-1 までででなく，N//2まで調べればいい。
- 2と3以上の奇数で割り切れるかだけ調べればいい。

これで計算量はおよそ半分になるのだが，計算量をオーダー記法で表すと$O(N)$のままである。しかし次のプログラムなら計算量は$O(\sqrt{N})$になる。というのもi * i <= Nというのはiが$\sqrt{N}$以下ということだからだ。

素数：例2

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

実は2002年にインド工科大学の研究者から出された『PRIMES is in P』という論文で，素数判定は$\log N$の多項式のオーダーでできることが証明されている。$O(N)$や$O(\sqrt{N})$は$\log N$の指数関数のオーダーであるから，これは非常に画期的な結果であった。もっとも，これで素数判定が速くなったということはないのだが（多項式の次数や係数が大きすぎるため）。

ところで，ここまでにあげたアルゴリズムは素因数を見つけることで素数判定をしているから，実質的には素因数分解に近いことをしていると考えられる。実は整数の性質を用いることで，具体的な素因数を見つけなくても素数かどうかを判定することができる。

たとえばフェルマーの小定理「**pが素数ならば，$2 \leqq a \leqq p-1$の範囲のすべての整数aに対して$a^{p-1} \equiv 1 \pmod{p}$**」の対偶「**$2 \leqq a \leqq p-1$の範囲のある整数$a$に対して$a^{p-1} \not\equiv 1 \pmod{p}$であれば，$p$は素数でない**」を用いて「**$2 \leqq a \leqq p-1$の範囲のすべての整数$a$に対して$a^{p-1} \equiv 1 \pmod{p}$ならば$p$は素数（だろう）**」とする判定方法を「フェルマーテスト」という。これは非常に簡便な方法ではあるのだが，合成数であってもこのテストを通ってしまう数（カーマイケル数）があるので，フェルマーテストで素数と判定されても確実にpが素数だとは言えない（そもそもフェルマーの小定理の逆だから，成り立つ保証がない。ちなみに合成数と判断されたものは確実に合成数）。

このように，合成数の判定は正しいが素数の判定が保証できない判定方法を「確率的素数判定」という（Miller-Rabin法なども有名）。ある程度の誤判定はあるのだが，実用的には差し支えないと考えて使うこともある。一方，確実に素数の判定ができるものを「決定的素数判定」というが，大抵は計算量が大きい。

3学期の情報の授業で「RSA暗号」について学習するが，これは素因数分解の計算量が大きいことを利用した暗号である。他にも離散対数や楕円曲線（といっても楕円ではない）が用いられる暗号もあるが，これらに共通するのは「一方向性」すなわち「**xから$f(x)$を求めることは簡単であるが，$f(x)$からxを求めることは極めて困難である**」ことである。このあたりは「$\mathrm{P} \neq \mathrm{NP}$問題」や「リーマン予想」と関係する極めて数学的な話であるが，量子コンピュータでどうこうできる話かもしれない。

3.2 数当てゲーム

1からNまでの整数をランダムに発生させ，少ない質問回数（ただし質問はYes/Noで答えられるもの）でそれを当てるゲームを考える。作戦として次の3つを考えてみたが，どれが効率がいいだろうか。

1. 当てずっぽうに数を言って，偶然当たったら終了。
2. 1から順に数を言って，当たったら終了。
3. a = 1，b = Nとし，次の操作を繰り返す（当たりをxとする）：
   (a) a == bならこれが当たりだから終了。
   (b) c = (a + b) // 2とする。x <= cならb = c，そうでなければa = c + 1とする。

偶然の要素を均すために，100回繰り返すプログラムにしたのが下の例である（random.randint(1, N)は1以上N以下の整数の乱数を生成する関数）。

数当てゲーム：例1

```
def f(N):
    count = 0
    for i in range(100):
        x = random.randint(1, N)
        count += 1
        while x != random.randint(1, N):
            count += 1
    return count
```

数当てゲーム：例2

```
def f(N):
    count = 0
    for i in range(100):
        x = random.randint(1, N)
        a = 1
        count += 1
        while a != x:
            a += 1
            count += 1
    return count
```

数当てゲーム：例3

```
def f(N):
    count = 0
    for i in range(100):
        x = random.randint(1, N)
        a = 1
        b = N
        while a != b:
            c = (a + b) // 2
            count += 1
            if x <= c:
                b = c
            else:
                a = c + 1
    return count
```

前にやったのと同じようにNの値によって実行時間がどう変化するかを表にまとめなさい。

N	例1	例2	例3
100			
1000			
10000			
100000			

例1，例2はともに$O(N)$であるが，例2の方が速い（例1は同じ数を何度もたずねてしまう可能性がある）。一方，例3はそれよりずっと実行時間が小さいはずだ。このことについて考察してみよう。

例3の考え方は，a<=x<=bを保ちながらaとbの間隔を半分にしていこうということだ。つまりk回で当たりに行き着いたとしたら，半分にするのをk回することで候補を1個に絞れるということだから，およそ$2^k = N$となる。このとき$k = \log_2 N$なので，例3の計算量は$O(\log N)$である。このような探し方を**二分探索**という。順番通りに探していく**線形探索**に比べればずっと速いのだが，あらかじめデータを大小順に並べておく必要があるので，そのための計算量が必要になることもある。

3.3 フィボナッチ数列

$1, 1, 2, 3, 5, 8, 13, \cdots$というように，前2つの項を足した値が次の項になるような数列を**フィボナッチ数列**という。ここでは関数を使ってこれを実現してみる。

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

具体的なプログラムは上の定義をそのまま書くことで実現できる。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。たとえば$f(4)$であれば$f(3) + f(2)$，その$f(3)$を$4(2) + f(1)$に，$f(2)$を$f(1)+f(0)$に…という風に遡って計算していく。

フィボナッチ数列：例1

```
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表にまとめてみよう。ただし例1は40まででやめておいた方がいい。後述するが，それ以上はあまりにも時間がかかりすぎる。実際，筆者のPCでは`f(50)`が30分くらい，`f(100)`に至っては100万年以上かかる計算になる（かなり大雑把な計算ではあるが）。

N	例1	例2	例3
10			
20			
30			
40			
50	×		
100	×		
1000	×		

たったの40でどうしてこんなに時間がかかってしまうのだろう。それを考えるために，例として$f(5) = 8$の計算を分解してみる。

1. $f(5) = f(4) + f(3)$
2. 上の式で$f(4) = f(3) + f(2), f(3) = f(2) + f(1)$であるから
   $f(5) = f(3) + f(2) + f(2) + f(1)$
3. さらに$f(2) = f(1) + f(0)$であるから
   $f(5) = f(2) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1)$
4. $f(5) = f(1) + f(0) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1)$

4回の作業で終わったように見えるが，1.では1箇所，2.では2箇所，3.では3箇所，4.では1箇所の計算をしているので，全部で7回の変換をしている。具体的は値がわかっているのは$f(0) = 1$と$f(1) = 1$だけなのだから，式全体が$f(1)$と$f(0)$だけに分解されるまで変換がなされているわけで，この例では$f(5)$が8個の$f(0)$，$f(1)$に変換されるということは$f(5) - 1 = 7$回の変換をしているのだ。$f(40) = 165580141$だから，これを求めるには165580140回の計算が必要，まして$f(50) = 12586269025$であるから…。

ではフィボナッチ数列の計算を簡単にやる方法はないのだろうか。そこで2つの方法を紹介しておく（どちらもPython特有の機能を使っている）。

フィボナッチ数列：例2

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

フィボナッチ数列：例3

```
def f(N):
    if N not in f.memo:
        f.memo[N] = f(N - 1) + f(N - 2)
    return f.memo[N]
f.memo = {0:1, 1:1}
```

例2は最初に書いた$1, 1, 2, 3, 5, 8, 13, \cdots$の項をループで求めている。$f(N)$は$f(1)$から番号を$N-1$増やせばいいので，繰り返し回数は$N-1$となる。

例3は既に計算した値を覚えておくというものだ。Pythonの「辞書型」といわれるデータ型の`f.memo`に`f.memo[0] = 1`，`f.memo[1] = 1`をメモしておいて，`N`が`f.memo`になかったら，値を計算して`f.memo[N]`にメモを追加している。

計算量は例2，例3ともに$O(N)$ではあるが，例3については既に計算した値を呼び出すときには$O(1)$である（と書いてある記事がネットには多いのだが，キーを探すためにはNに応じた計算量–よくある実装だと$O(\log N)$くらい–が必要ではないのかなあ）。ちなみに例1はおよそ$O(1.62^N)$なのでNが10増えると計算量は約120倍になる。

第4章　おわりに

コンピュータの性能が低かった時代は，少しでも（時間・空間ともに）計算量を減らそうと工夫をした。データの使いまわしはもちろん，プログラムの自己書き換えさえもよく使われる手段の一つであった。そこまでいかなくても，たとえば本校の成績処理システムでも時間がかかる集計の結果は別途保管しておいて，何度も計算しないで済むようにしていた。

しかし今は僅かな計算量の節約よりも，開発・運用の簡便さが求められている。先の成績処理システムの例も，当時は得点修正のたびに集計作業が必要で，それを忘れると不整合が生じたのだが，ある時期から集計は必要になるたびにその場で行うように変更した。コンピュータの速度改善がそれを可能にしたのだ。こういったやり方は「富豪的プログラミング」と揶揄されたりもしたが，逆にいえばそれができるようにコンピュータは進化してきた。

とはいえ，細かい改良が必要な世界は歴としてあるのだし，そこまでいかなくてもフィボナッチ数列を100項求める程度のことで困難が起きるようでは話にならない。もっと効率のいい処理方法を求めることは常に求められることである。そしてそれはコンピュータに関するものだけではないだろう。我々の生活や仕事についても，社会や…おや，誰かが来たようだ。