アルゴリズムと計算量オーダー

アルゴリズムとは「やり方」とか「手順」とかそんな感じの言葉で、計算量というのは何かしら処理を実行するときにかかる時間とかコストという意味合いで使う。

本当は計算量というと、処理にかかる時間だけではなく、処理で使用するメモリなどの空間量でも使われる言葉らしい。でもここでは時間的な量として計算量という言葉を使う。

計算量

計算量といっても具体的にこの計算には何秒かかるとか計算するわけではなく、概ねこんなもんだろうというどんぶり勘定で話が進む事が多いようだ

なのでここでもそういう感じの扱いとして、具体的に計算量のイメージを固めていく。

例えば足し算にかかる時間を**1**とすると

```
int sum = 0;
for(int i = 0; i < 100; ++i) {
    sum += i;
}
```

この処理では`sum+=i`という足し算を100回しているので、計算量は**100**ということになる。

厳密にはループ内の条件判定であったり、インクリメントや代入というのも計算だし、時間もかかるので計算量に加えてやるのが正しい見方だと思うが、計算量の話題でそこまで厳密に考えることはなさそうである。

次はこの処理である

```
int ret = 1;
for(int i = 0; i < 100; ++i) {
    ret *= i;
}
```

掛け算は足し算より時間のかかる処理なので、掛け算にかかる時間を**10**とする。

掛け算を100回行っているので、計算量は**100×10**で**1000**という事になる。

ただ実際に足し算や掛け算の処理にどれくらいの時間がかかるのかというのは環境によるわけで、アルゴリズムの計算量を求める場合はやはりこういう細かい違いは無視してしまう事が多いようだ。

最終的に気にしているのは**処理の回数**であって、計算量という言葉も大雑把に言えば**何回処理をするのか**という感覚で使われている。

なので、ここでもそこらへんの細かい事は無視して足し算も掛け算も時間的には**1**ってことにするし、細かい事は気にするな精神で進めていくし、後半になるほどもっと大雑把に捉える事になっていく。

計算量と時間のイメージ

ここに1GHzのCPUがあったとする。

1GHz = 約1000M = 約 1,000,000K = 約 1,000,000,000 = 約10億であり、このCPUは1秒間に約10個の命令を処理できるCPUという事である。

足し算・掛け算が1命令で実行できるとした場合

```
double ret = 0;
for(int i = 0; i < 10000; ++i) {
    for(int j = 0; j < 10000; ++j) {
        ret += i * 10000 + j;
    }
}
```

この処理は10000x10000で1億回ループしていて、その中で足し算と掛け算を1回ずつしている、つまり

足し算：1億回

掛け算：1億回

となって、この処理は2億個の命令が実行される事になる。

1GHzのCPUでは、1秒間に10億個の命令が処理できるので、この処理にはだいたい0.2秒かかるという計算だ。

実際は厳密に何秒かかるかは動かしてみないとわからないが、計算量が分かれば、少なくともこの処理は重いとか、そんなに重くならないなという事が予め予測できるようになるのである。

もう少し複雑な計算量

計算量についてもう少し具体的に見ていくために、以下の多項式を計算するプログラムを考えてみる。

$$ax^3 + bx^2 + cx + d$$

とりあえず愚直に関数にしてみる、係数やxは引数でもらう事にする。

```
int y(int x, int a, int b, int c, int d)
{
    int y, t;

    // ax^3
    t = a;
    t *= x;
    t *= x;
    t *= x;
    y = t;  // ax^3

    // bx^2
    t = b;
    t *= x;
    t *= x;
    y += t; // ax^3 + bx^2

    // cx
    t = c;
    t *= x;
    y += t; // ax^3 + bx^2 + cx

    // d
    t = d;
    y += t; // ax^4 + bx^2 + cx + d

    return y;
}
```

この処理の足し算と掛け算の回数に着目すると

- 足し算：4回
- 掛け算：$3+2+1+0$で6回

この関数の計算量は足し算4回、掛け算6回で、まぁだいたい**10**ということである。

これは項が4つしかないので、手計算でも計算量を求める事はできたが、では項が5つ、6つと増えていくとしたらどうだろうか？

仮に項数が100万になった場合、計算量はどうなるのか、現実的な時間で終わる処理なのか？

このように扱う項数(データ数)が増えた場合にどうなるかを考える指標として計算量オーダーがあり、今からこの処理の計算量オーダーを求めてみたいと思う。

項数がn個の処理に修正する

項が増える度に係数a,b,c,d,e...と増えていくし、文字も足りなくなるし融通がきかないので、係数は配列にまとめてしまう。

$$a_3x^3 + a_2x^2 + a_1x + a_0$$

数式で表すと添え字まみれで見づらいが、プログラムにすればスッキリするので我慢だ。

またプログラムでループする場合は、ループカウンタが0から増えていく方が見やすい(個人的に)ので、式も添え字の小さいもの順に並び変えた(足し算は順番を変えても答えは変わらない)

$$a_0 + a_1x + a_2x^2 + a_3x^3$$

項がn個の場合なので、数式はこんな感じになる。(nは0から数えてるので末項はn-1)

$$a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_{n-1}x^{n-1}$$

数学ではこういう数の総和はシグマ記号を使って表すことが多いので、シグマで書くとこんな感じだ。

$$\sum_{i=0}^{n-1} a_i x^i$$

これをプログラムにするとこうなる

```
// n  : 項数
// x  : 変数
// a[]: 係数
int y(int n, int x, int a[])
{
    int y = 0;

    // 足し算の数(項数)だけループ
    for (int i = 0; i < n; ++i)
    {
        int t = a[i];

        // 掛け算の数だけループ
        for (int j = 0; j < i; ++j)
        {
            t *= x;
        }
        y += t;
    }

    return y;
}
```

計算量を考える

この処理の計算量を、足し算と掛け算の回数として考えることにすると、それぞれの回数は以下のようになる。

足し算：n回

掛け算：0+1+2+...+n-1回

数式を見ても、足し算は項の数だけあるし、掛け算はx, x^2, x^3とだんだん増えていくのがわかる。

$$a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_{n-1}x^{n-1}$$

掛け算の数は0から$n-1$までの数の和だが、どんぶり勘定で1からnまでの和と考えると、これは公式があって以下の式で求められる。

$$1\text{から}n\text{までの和} = \frac{n(n+1)}{2}$$

この処理の計算量は足し算と掛け算の回数の和なので、こうなる。

$$\text{足し算の回数} + \text{掛け算の回数} = n + \frac{n(n+1)}{2}$$

具体的な数値を入れてみる

nが100だとしたら

$$100 + \frac{100 \cdot 101}{2} = 100 + 5050$$

計算量は5150ということになる

nが100万だとしたら

$$100\text{万} + \frac{100\text{万} \cdot 101\text{万}}{2} = 100\text{万} + 5050\text{億}$$

計算量は5050億100万、まぁだいたい5000億くらいで、1GHzのCPUであれば、500秒くらいかかる計算になる。

計算量オーダー

計算量オーダーというのは計算量を求める式の次数である。

例えばこれは先ほどの関数の計算量を求めた式である。

$$n + \frac{n(n+1)}{2}$$

この式を展開して括弧を外すとnの2次式になる

$$n + \frac{1}{2}n^2 + \frac{1}{2}n$$

計算量オーダーを考えるときは、計算量を求める式の一番高い次数の項に着目する、つまり赤文字にした項だけ見る。

$$n + \frac{1}{2}\color{red}{n^2} + \frac{1}{2}n$$

さらに、計算量オーダーでは係数も無視する

$$n + \frac{1}{2}\color{red}{n^2} + \frac{1}{2}n$$

つまりn^2の部分だけに着目する。

今回の関数の計算量オーダーはn^2という事になる。

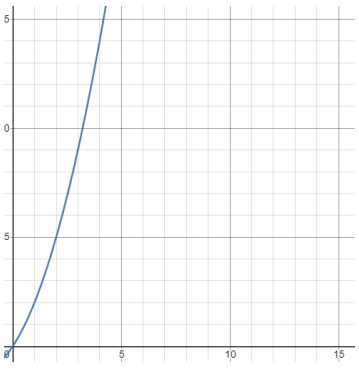
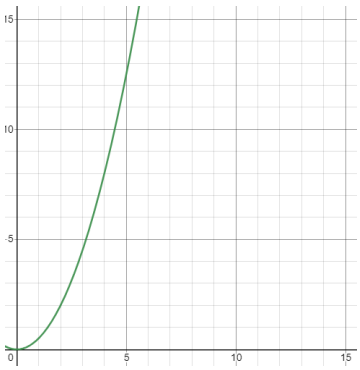
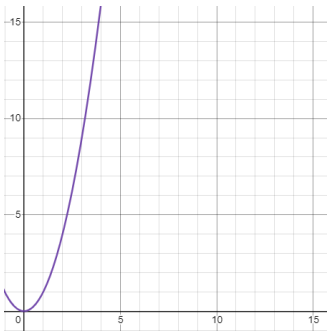
また最初は足し算や掛け算の数を数えていたが、計算量オーダーは**ループする数**と言っても差し支えない。

nこのデータを処理する時に1回のループで済むなら$O(N)$だし、2重ループになるなら$O(N^2)$とも言える。

係数や他の項を無視していいの？

一番次数が高い項以外を無視したり、係数をなくしても大丈夫なのかというと、基本的に問題ない。

nが大きくなっていくと次数が低い項や係数は誤差になってくる。参考までに上の式のグラフを並べてみる。

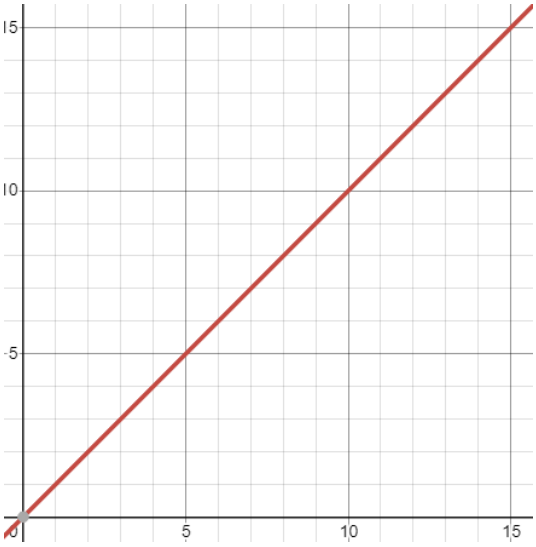
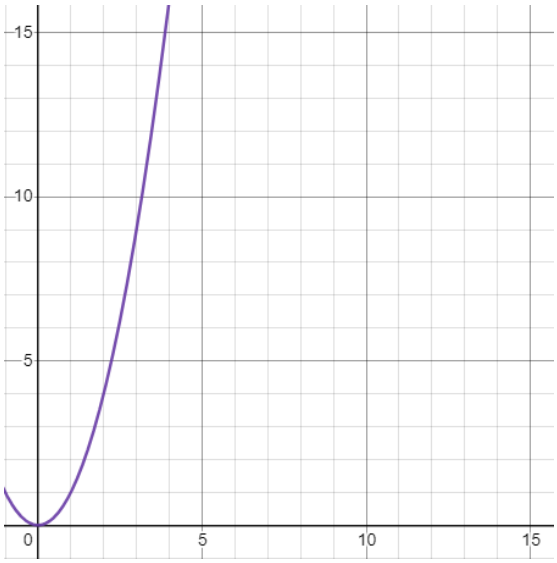
$n + \frac{1}{2}n^2 + \frac{1}{2}n$	$\frac{1}{2}n^2$	n^2
		

多少の違いはあれど、どのグラフもさほど形は変わらないし、nが増えた時の数値の増え方も同じようなものである。

なので計算量オーダーでは最も次数が高い項だけに着目すればよい。

計算量オーダーでわかること

以下にnとn^2のグラフを用意した。

計算量オーダー n	計算量オーダー n^2
	

計算量オーダーがnの場合はnに比例して計算量も増えていくが、n^2の方はnが増えると計算量はn^2のペースで増えていくという大まかな未来がわかる。

このように計算量オーダーがわかればデータが増えた時に、どういうペースで計算量が増えていくかがわかる。

ビッグオー記法

計算量オーダーの表し方として大文字のOを使って表す方法がある。

今まで計算量オーダーはn^2と言っていたものは、ビッグオー記法で書くと以下のようになる。

$$O(N^2)$$

今回実装したアルゴリズムの計算量オーダーは$O(N^2)$であるというのは、Nが増えると計算量がN^2で増えていくアルゴリズムだよという意味になる。

アルゴリズムの威力

今回実装したアルゴリズムは$O(N^2)$であったが、これはあまり効率の良いアルゴリズムではない。

$O(N^2)$というのはNが10000になれば、計算量は1億になってしまうし、プログラムで10000件のデータを扱うことは珍しくない。

なので今回は以下の$O(N^2)$の処理を、アルゴリズムを工夫することで$O(N)$にしてみようと思う。

```
// n  : 項数
// x  : 変数
// a[]: 係数
int y(int n, int x, int a[])
{
    int y = 0;

    // 足し算の数(項数)だけループ
    for (int i = 0; i < n; ++i)
    {
        int t = a[i];

        // 掛け算の数だけループ
        for (int j = 0; j < i; ++j)
        {
            t *= x;
        }
        y += t;
    }

    return y;
}
```

この処理は元々以下の計算をするアルゴリズムであった。

$$ax^3 + bx^2 + cx + d$$

この式を因数分解して、以下のように表してみる。

$$((ax + b)x + c)x + d$$

これを愚直にプログラムにしてみるとこうなる。

```
int y = 0;
y += a; // a
y *= x; // ax
y += b; // ax + b
y *= x; // (ax + b)x
y += c; // (ax + b)x + c
y *= x; // ((ax + b)x + c)x
y += d; // ((ax + b)x) + c)x + d
```

項数がn個に対応できるようにしたものがこれになる。

```
// n  : 項数
// x  : 変数
// a[]: 係数
int y(int n, int x, int a[])
{
    int y = 0;
```

```
    for (int i = 0; i < n - 1; ++i) {
        y += a[i];
        y *= x;
    }

    y += a[n - 1];

    return y;
}
```

足し算の回数はn+1回、掛け算の回数はn回になるので計算量は

$$(n+1)+n=2n+1$$

になる。

これはnの1次式なので、計算量オーダーは$O(N)$になる。

同じ事をする処理でもアルゴリズムを改善することで、計算量を減らせる例である。

また、処理速度ギリギリをせめる場合はその限りではないだろうが、ちょっとアルゴリズムを改善したところで、計算量オーダーが変わらないのであれば、そこまでの改善は見込めないとも言える。