アルゴリズム入門

アルゴリズムとは？

　アルゴリズムとは，簡単に言ってしまえば「コンピュータを用いた効率的な問題の解き方」です．近年でも様々な発展があります．
　アルゴリズムというのは楽しいものです．その楽しさを伝えることができたら本望です！．ぜひ，あなたもアルゴリズムマニアの道に足を踏み入れて下さい．
　アルゴリズムマニアに必要なものは何かと聞かれたら！．私は次の3つの力だと思います．

1. いくらかの数学力
2. 読めるだけでOKだから英語力
3. 猫のような好奇心

　数学力と言っても，著者も大してありません．でも，社会人になってからも地道に勉強を続けて，多少は進歩しています．数学という言語は，コンピュータ言語にも通ずる楽しく苦しい？！言語です．数学が完璧に分かれば楽しいだろうになぁと思うのです．
　また，やはり日本語だけだと情報が狭い気がしています．コンピュータアルゴリズム関連の書籍の英語は簡単です．ぜひとも食わず嫌いにならずにチャレンジしてみてください．知識の海で泳ぐのは楽しいですよ～～．

　最後はやっぱり好奇心．新しいものとか，奇妙なものとか，楽しい物事にひかれる柔らかい心の持ちようが大事だと思います．

アルゴリズムの実行時間について

　アルゴリズムの実行時間の目安として，O(？)という表記を使います．まず，この表記法の意味について説明します．
　この表記の意味が理解できれば，高速なアルゴリズムとは？，また，高速なアルゴリズムを書くにはどうしたらよいか？，という問いに答えることができるようになります．

　まず，実行時間とデータ量nの関係についてO(？)表記を使う場合について説明します．

　O(1)というのは，定数時間で実行が終わるものを表しています．データ量がいくらあろうとも実行時間とは無関係な場合です！．

　O(log n)というのは，データ数nの対数時間に比例して実行が終わるものを意味しています．これもかなり高速です．例えば，探索領域を2分割してどちらかをさらに調べればよいかを確定し，そちらをさらに2分割，というようなアルゴリズムでは，O(log n)になります．

　O(n)というのは，データ数nに比例した時間で実行が終わるものです．データがあって単純に先頭から最後まで探索するようなアルゴリズムがO(n)です．

　O(n log n)というのは，例えばデータを先頭から順番に調べていき，個々のデータに対してO(log n)の処理時間がかかるようなアルゴリズムなどです．このくらいまでが，高速なアルゴリズム，ということができます．

　O(n^2)以上になってくると，多数のデータがあるとき対処できなくなってきます．重いアルゴリズムと言うことができます．

　さて，具体例で説明します．まず，O(n)の場合は1重ループです．

```
for (int i=0; i<n; ++i ) {
　hogehoge～～～
}
```

　このループはO(n)ということができます．

O(n^2)も簡単で2重ループの場合です．

```
for (int i=0; i<n; ++i ) {
   for( int j=0; j<n; ++j ) {
   hogehoge～～～
   }
}
```

さて，ここで問題です．次のプログラムのO(?)はいくつでしょうか？

```
for(int i=0; i<10000000; ++i ) {
   hogehoge～～
}
```

　う～ん．1000000回もループしているからオーダーも大きいに違いない？と思ったら間違いです．

　入力データ数nとの関係から見ると，nがいくら増えても実行時間は一定，すなわち定数です．したがってO(1)ということになります．

　次の問題です．

```
for(int i=0; i<n/2; ++i ) {
   hogehoge～～
}
```

　次のように考えます．ループ回数がn/2だから 1/2 nに比例した計算時間がかかります．係数部分を無視して，O(n)となります．

　さて，次のもうちょっと複雑な問題です．

```
for(int i=0; i<n/2; ++i ) {
   for(int j=0; j<n/100; ++j ) {
     hogehoge～～
```

```
    }
}
```

まず，ループ回数が 1/2 n × 1/100 n = 1 / 200 n^2 となります．係数部分を無視すると，O(n^2)となります．

次にO(log n)の場合です．どんなプログラムが該当すると思われますか？

```
for(int i=n; i>=0; i/=2 ) {
  hogehoge〜〜
}

for(int i=0; i<n; i*=2 ) {
  hogehoge〜〜
}
```

このようなプログラムがO(log n)となります．定数2の部分は任意の数でOKです．本質をつかんで下さい．

O(n log n)などは上記の単純な応用なので省略します．

アルゴリズムの実行時間について 2

では，nと実行時間の関係について表にしてみます．

	log n	n	n log n	n^2
10	1	10	10	100
100	2	100	200	10000
1000	3	1000	3000	1000000
10000	4	10000	40000	1E+08

どうでしょうか？　log nが高速, nとn log nが同じくらい, n2が少し離れていて，だいたい3つのグループに分かれることがわかると思います．また，n2の実行時間が入力データ数に従って急速に膨れ上がるのもわかると思います．

このように実行時間の大まかな見積りができることがO(？)表記のメリットです．以後，この表記を用いてアルゴリズムの説明を行います．

プログラムを見て，ややこしいことをしているなぁ～と思うことがあると思います．でも，なぜか高速．．．そういった場合はこの計算量の考え方で考えれば，高速な秘密がわかります．プログラムは一見複雑でもO(1)とか，O(n)とかの部分が長くなっていて，O(n?)などの部分がない，もしくは，低く抑えられているから，高速なんです．これが理解できれば，自分で本質的に高速なプログラムを書く事もできるようになります！．

力技アルゴリズム

特に効率を考慮せずとも力技で解ける問題があります．力技，というのは，すべての場合を調べつくす，とか，それを少し改良した程度のやり方を意味します．
力技が有効なのは，主に次のような場合です．

1. 一回しか解かないような問題
2. 問題の様子見のため
3. 力技でも許容時間内に解けるような問題
4. 他の複雑な技法の計算結果と比較して，正しいことを確認するため

一回しか解かない問題に対して複雑なアルゴリズムを適用すると，投入したプログラムの労力に見合うかどうかが疑問です．力技で許容できる時間内で解けるならば，力技で解いてしまうのがよいと思います．

解きたい問題について知識が少ないとき，とりあえず解の様子を探りたい場合があります．そのようなときに，力技で解いてみることは有効な手段です．また，この試みにより，力技で解けない問題であることが判明するかもしれません．

問題を解く速度が重要でない場合や，力技で十分に解ける規模の問題のときは，力技で解くのがよいと思います．問題を必要以上に複雑にしないことです．

最後に，力技で解くプログラムがあると，その計算結果を用いて自動無限テストを作ることができます．自動無限テストとは，乱数で入力データを生成し，その入力値で計算して，その結果が正しいかをチェックし，間違っていなければこのテストをずっと繰り返すという手法です．この手法で，結果が正しいかどうかをチェックする際に，力技プログラムが役に立ちます．自動無限テストについては，のちの章で詳しく解説いたします．

再帰について

　再帰とは，関数が自分自身を呼び出す手法のことです．というと，判然としないと思いますので，次に簡単なプログラムで考え方を示します．

プログラムリスト２１　Recursive.cpp

```
#include <cstdio>
int Fibonachi( int nVal ) {

   if ( nVal <= 1 )
      return 1;

   return Fibonachi( nVal-1 ) + Fibonachi( nVal-2 );
}

int main() {

   for(int nVal=3; nVal<=10; ++nVal ) {
      int nAns = Fibonachi( nVal );
      printf("f(%d) = %d\n", nVal, nAns );
   }
   return 0;
}
```

プログラムの出力は次のようになります．

```
f(3) = 3
f(4) = 5
f(5) = 8
f(6) = 13
f(7) = 21
f(8) = 34
```

f(9) = 55
f(10) = 89

　さて，プログラムをじっくりと見てみて下さい．実用的なプログラムではないですが，再帰，という概念を示すもの，ということで，その点は無視して下さい．のちのアルゴリズムの説明で，再帰を頻繁に用いますが，それに慣れておくための説明用です．
　関数Fibonachiの中で，自分自身を呼び出していますね？．これが再帰の考え方です．
　再帰のポイントは，次の3つです．
１．　必ず再帰呼び出しが停止するように作ること．さもないとメモリを使い尽くすまで止まりません．
２．　余分な再帰呼び出しはなるべく避けるように作ること．そうすると速度が大幅にアップします．
３．　再帰呼び出しが自然に当てはまる問題と，そうでない問題を見分けること．上の例は，再帰呼び出しが不自然な例です．平書きにした方が速度の点でもわかりやすさの点でも上回ります．

　再帰については，以下のサンプルでも使っていくので，徐々に慣れていってください．