アルゴリズム

大熊　俊明

最終更新 2012/12/26

1　はじめに

未完のアルゴリズム入門本です．これからガシガシと書いていきます．

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．

A.　いくらかの数学力

B.　読めるだけで OK だから英語力

C.　猫のような好奇心と考えることが好き

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．そして飽きずに考えられる力が重要であると思います．

1.1 プログラムの実行環境について

本書のプログラムは，C++を使って書かれています．本書では特に cygwin の g++を使ってチェックしてあります．

Cygwin は http://www.cygwin.com から入手できます．

1.2 捧げる

我が甥っ子の秀斗くん，蒼介くん，そして，姪の芹葉ちゃんにこの文書を捧げます．みんなが大きくなったとき，この文書を読んでもらえるように頑張って書きます．

2　基礎的なアルゴリズム

2.1 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るようです．

2.2 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．

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，一般に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．一般に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．

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， n^2 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，n^2 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，$O(n^2)$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！．

2.3 ユニットテストについて

ユニットテストとは，関数単位などで行う小さなテストのことです．特に計算プログラムを書く際に有用です．本書のプログラムでは，チョコチョコとユニットテストを書いていきます．筆者は通常は Google C++ Testing Framework を使っていますが，本書では簡易性を優先して標準ライブラリを使用して簡単にチェックします．

ユニットテストは，関数やクラスの仕様を示す，という役割もあります．

ユニットテストが有効なプログラムでは，ぜひともユニットテストを書いていきましょう．作業時間が大幅に短縮できることが経験的に多いです．

2.4 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. 力技でも許容時間内に解けるような問題
3. 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．

2.5 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 2-1　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 Fibonachi 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.

2.5.1　課題：

ウォーミングアップです．上記の Recursive.cpp のプログラムをベタ書きのプログラムに直してみてください．ベタ書き，というのは，再帰を使わない書き方のことです.

2.5.2　解答：

解答プログラムです．こうして解答プログラムを書いて見ると，再帰を使うと簡単に書けるなと実感するかもしれません.

```
#include <cstdio>
```

```
int Fibonachi( int nVal ) {

  if ( nVal <= 1 )
    return 1;

  int nBef[2] = { 1, 2 };

  for( int it=3; it<=nVal; ++it ) {

    int nSum = nBef[0] + nBef[1];

    // 今の数を 1 つ前に
    // 1 つ前を 2 つ前に
    nBef[0] = nBef[1];
    nBef[1] = nSum;
  }
  return nBef[1];
}

int main() {

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

2.6 リスト構造

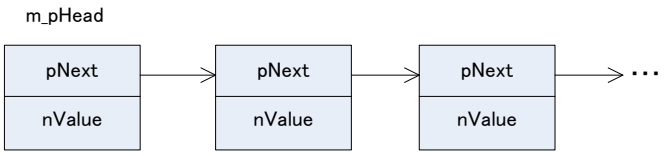
リスト構造とは，一言で言えば，動的メモリ確保を用いてメモリを確保しながら，値を記録するための配列に似たイメージのデータ構造のことです．C++でも C#でも既存のライブラリを使えば，簡単なデータに対しては自分でリスト構造を用意する必要はなくなりましたが，しかし，より複雑なデータ構造を実現するときや上級のアルゴリズムを使う際には，必須の知識と言えますので，ここに簡単にご紹介します．

リスト構造と配列の違いとして，配列はあらかじめ大きさが決まっていますが，リスト構造はどんどん動的にメモリを確保していくので，上限をあらかじめ決めておく必要がないということがあります．また，配列のアクセスは O(1)ですが，リスト構造の場合は工夫をしない限り先頭からデータを順番にたどっていくので，O(n)となります．

2.6.1　課題

では，以下のプログラムリストを実行したときに，正常に実行されるように，class List を実装してみてください．

分からないようならば図を考えて動きを理解して下さい．また，各メソッドの使い方は，ユニットテストをみて理解して下さい．



データ構造は，上図のようになっています．

リスト構造は他人のプログラムを眺めるだけでなく，一度，自分で実装されることをおすすめします．慣れていないと，けっこう，ハマるものですし，楽しいですよ！．

プログラムリスト 2-2　List_Kadai.cpp

```
#include <cstdio>
#include <cassert>

// リストのノード（これを利用してください）
struct ListNode {
    ListNode    *pNext;
    int         nValue;
    ListNode() : pNext(0) {}
    ListNode( ListNode *pNext_, int nValue_ ) : pNext(pNext_), nValue(nValue_) {}
    ~ListNode() { delete pNext; }
};

// 課題：以下のクラスを実装する
class List {
public:
    List() {}
    ~List() {}

    // データを加える
    void    AddData(int nValue) {}

    // データを削除する
    bool    DeleteData(int nIndex) { return true; }

    // サイズを得る
    int Size() const { return 0; }

    // 値を得る
    int GetData(int nIndex) const { return 0; }
};
```

```
// ユニットテスト
// 仕様書でもあります
void RunUnitTest() {
    {
        List listTest;
        assert( listTest.Size() == 0 );
        listTest.AddData( 10 );
        assert( listTest.Size() == 1 );
        listTest.AddData( 20 );
        assert( listTest.Size() == 2 );

        assert( listTest.GetData( 0 ) == 20 );
        assert( listTest.GetData( 1 ) == 10 );

        listTest.DeleteData(0);
        assert( listTest.Size() == 1 );
        assert( listTest.GetData( 0 ) == 10 );

        listTest.DeleteData(0);
        assert( listTest.Size() == 0 );
    }
}

int main() {
  RunUnitTest();
  return 0;
}
```

2.6.2　解答例

それでは解答例のプログラムです.

プログラムリスト 2-3　List.cpp

```
#include <cstdio>
#include <cassert>

// リストのノード
struct ListNode {
    ListNode    *pNext;
    int         nValue;
    ListNode() : pNext(0) {}
    ListNode( ListNode *pNext_, int nValue_ ) : pNext(pNext_), nValue(nValue_) {}
    ~ListNode() { delete pNext; }
};

// リスト本体
// 効率は無視してます.
class List {
    ListNode *m_pHead;
```

```
public:
    List() : m_pHead(0) {}
    ~List() { delete m_pHead; }

    // データを加える
    void    AddData(int nValue) {
        if( m_pHead ) {
            ListNode *pNewNode = new ListNode( m_pHead, nValue );
            m_pHead = pNewNode;
        }
        else {
            m_pHead = new ListNode( 0, nValue );
        }
    }

    // データを削除する
    bool    DeleteData(int nIndex) {

        ListNode *pBefNode = m_pHead, *pCurNode = m_pHead;
        for( ; pCurNode && nIndex > 0 ; --nIndex ) {
            pBefNode = pCurNode;
            pCurNode = pCurNode->pNext;
        }

        if( pCurNode ) {

            if ( pCurNode == m_pHead )
                m_pHead = m_pHead->pNext;
            else
                pBefNode->pNext = pCurNode->pNext;

            pCurNode->pNext = 0;          // 連鎖して消えてしまわないように
            delete pCurNode;

            return true;
        }
        else
            return false;
    }

    // サイズを得る
    int Size() const {

        int nCnt = 0;
        for( ListNode *pCurNode = m_pHead; pCurNode; pCurNode = pCurNode->pNext ) {
            ++nCnt;
        }
        return nCnt;
    }

    // 値を得る
    int GetData(int nIndex) const {

        ListNode *pCurNode;
```

```
        for( pCurNode = m_pHead; pCurNode && nIndex > 0 ; --nIndex )
            pCurNode = pCurNode->pNext;

        if ( pCurNode )
            return pCurNode->nValue;
        else
            return 0;
    }
};

// ユニットテスト
// 仕様書でもあります
void RunUnitTest() {
    {
        List listTest;
        assert( listTest.Size() == 0 );
        listTest.AddData( 10 );
        assert( listTest.Size() == 1 );
        listTest.AddData( 20 );
        assert( listTest.Size() == 2 );

        assert( listTest.GetData( 0 ) == 20 );
        assert( listTest.GetData( 1 ) == 10 );

        listTest.DeleteData(0);
        assert( listTest.Size() == 1 );
        assert( listTest.GetData( 0 ) == 10 );

        listTest.DeleteData(0);
        assert( listTest.Size() == 0 );
    }
}

int main() {
    RunUnitTest();
    return 0;
}
```

リスト構造のメリットは，上限をあらかじめ指定しておく必要がないことや，必要なデータ数に比例した量しかメモリを消費しない点があります．また，挿入や削除が O(1)で済む，ということも特徴で，頻繁に挿入，削除が繰り返される場合には有用なデータ構造です．一方，欠点として，あらかじめデータ量がわかっているならば，メモリ効率が悪いことや，中間位置などの要素にアクセスするために O(n)の時間がかかることです．

C++の STL では list<>がここで紹介したリスト構造に対応するライブラリです．これらのクラスがどういった挙動をするのか，なぜそういった挙動をするのか，などが，ここでご紹介した内容を理解することで納得できると思います．そういった意味でも，リスト構造を理解しておく意味があります．

2.7 キューとスタック

キューとスタック，という古典的ではありますが，重要なデータ構造について紹介します．キューとは最初に入れたデータから順番に出てくるリスト構造，スタックとは最初に入れたデータが最後に出てくるデータ構造です．これらも C++では標準ライブラリに用意されていますので，単純なキューやスタックを使う際には，それらのライブラリを使えばよいですが，一応，動作機構を押さえるという意味でここに詳細にご紹介します．

2.7.1　課題

以下のユニットテストを通るようにキューとスタックのクラスを実装してください．

なお，実装にはポインタを用いて実装をすることもできますが std::list<>を用いても OK です．

プログラムリスト 2-4　StackAndQueue_Kadai.cpp

```
#include <cstdio>
#include <cassert>
#include <list>

// 以下のクラスを実装してください
class Stack {
  std::list<int>    m_listStack;
public:
  void    Push(int nData) {}
  int     Pop() { return 0; }
};

// 以下のクラスを実装してください
class Queue {
  std::list<int>    m_listQueue;
public:
  void    Enqueue( int nData ) {}
  int        Dequeue() { return 0; }
};

void RunUnitTest() {

  // Stack Test
  {
    int nTmp;
    Stack test;
    test.Push(1);
    nTmp = test.Pop();
    assert( nTmp == 1 );

    test.Push(2);
    test.Push(3);
    nTmp = test.Pop();
```

```
    assert( nTmp == 3 );
    nTmp = test.Pop();
    assert( nTmp == 2 );
  }
  // Queue Test
  {
    int nTmp;
    Queue test;
    test.Enqueue(1);
    nTmp = test.Dequeue();
    assert( nTmp == 1 );

    test.Enqueue(2);
    test.Enqueue(3);
    nTmp = test.Dequeue();
    assert( nTmp == 2 );
    nTmp = test.Dequeue();
    assert( nTmp == 3 );
  }
}

int main() {
  RunUnitTest();
  return 0;
}
```

2.7.2 解答例

プログラムリスト 2-5　StackAndQueue.cpp

```
#include <cstdio>
#include <cassert>
#include <list>

class Stack {
  std::list<int>    m_listStack;
public:
  void    Push(int nData) {
    m_listStack.push_back( nData );
  }
  int        Pop() {
    int nRet = m_listStack.back();
    m_listStack.pop_back();
    return nRet;
  }
};

class Queue {
```

```
  std::list<int>    m_listQueue;
public:
  void    Enqueue( int nData ) {
    m_listQueue.push_back( nData );
  }
  int        Dequeue() {
    int nRet = m_listQueue.front();
    m_listQueue.pop_front();
    return nRet;
  }
};

void RunUnitTest() {

  // Stack Test
  {
    int nTmp;
    Stack test;
    test.Push(1);
    nTmp = test.Pop();
    assert( nTmp == 1 );

    test.Push(2);
    test.Push(3);
    nTmp = test.Pop();
    assert( nTmp == 3 );
    nTmp = test.Pop();
    assert( nTmp == 2 );
  }
  // Queue Test
  {
    int nTmp;
    Queue test;
    test.Enqueue(1);
    nTmp = test.Dequeue();
    assert( nTmp == 1 );

    test.Enqueue(2);
    test.Enqueue(3);
    nTmp = test.Dequeue();
    assert( nTmp == 2 );
    nTmp = test.Dequeue();
    assert( nTmp == 3 );
  }
}

int main() {
  RunUnitTest();
  return 0;
}
```

　このスタックやキューは，リスト構造が基本となっています．リスト構造自体はたいしたことないかもしれませんが，様々な構造と組み合わせるような複雑な場合に意味が出てき

ます.

2.8　2 分木構造

木構造について説明します．木構造については，ライブラリでは map<>や set<>と言った木構造を内部で使っているものはありますが，直接サポートはされておらず，自前でポインタを用いて作る必要があります．また，のちに説明するデータ構造のための入門的な意味もあります.

２分木構造というのは，値を入れる際にツリー状に値を入れていくものです．と言葉で書いても分かりにくいと思うので，図で説明します.

まず，値を 1 ついれた 2 分木です.

さて，親よりも小さな値は左の子，大きな値は右の子に入れるとします.

（木構造のプログラム）

2.9 ヒープ

ヒープというのは，緩やかにソートされた２分木みたいなものです．最小（もしくは最大）の要素がすぐに求まる，という特徴があります．C++のライブラリには対応する関数が存在します.

2.10 優先順位付きキュー（プライオリティキュー）

優先順位付きキューというのは，各要素が優先順位を持っているキューのことです．要素を取り出す際に，最も優先順位の高い要素から取り出されていきます.

2.11 動的配列

動的配列の構造について示します．これは STL の std::vector<>の実装の簡略版でもあります．この実装により，std::vector<>がどういった仕組みなのか，また，std::vector<>に対して，どういった制約が課されているのか理解して下さい．この動的配列の実装をよく理解することで，std::vector<>の挙動が理解できます.

2.12 動的計画法

2.13 構文解析

3　浮動小数点数を用いた無誤差演算について

幾何アルゴリズムというのは. RAM モデルという丸め誤差がないという仮定の理想的で非現実的な計算モデルを使って計算がされます. そして，このようなアルゴリズムが現実で破綻しないようにするために，多くの労力が必要となります.

整数を用いた無誤差演算というのは一般的です.

4　許容誤差付き幾何演算について

許容誤差付き

5　３次元グラフィック

さて，何を題材に使用かな？

Ray Casting 法

OpenGL を対象とする.

6　最終章

楽しんでいただけましたか？^^.