プログラミング技法

第3回演習

資料は

http://www-hiraki.is.s.u-tokyo.ac.jp/lectures/prog_giho/

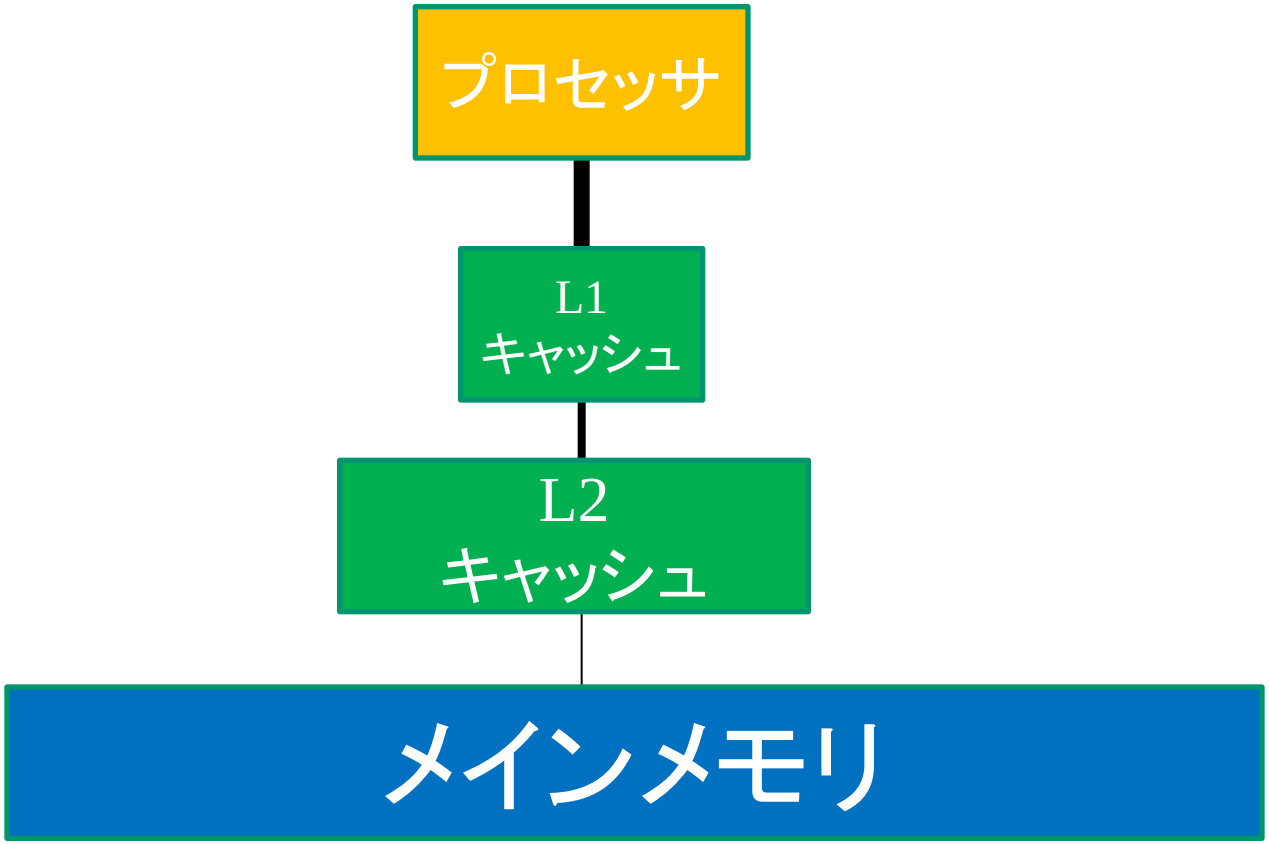
今日の目標

- 前回の続き、行列積まで実現する
- 実行時間の測り方になれる
- ３言語全部で行列積問題を書く（Optional)

- 3言語になれる例題を試してみる

何故実行時間が変わるか

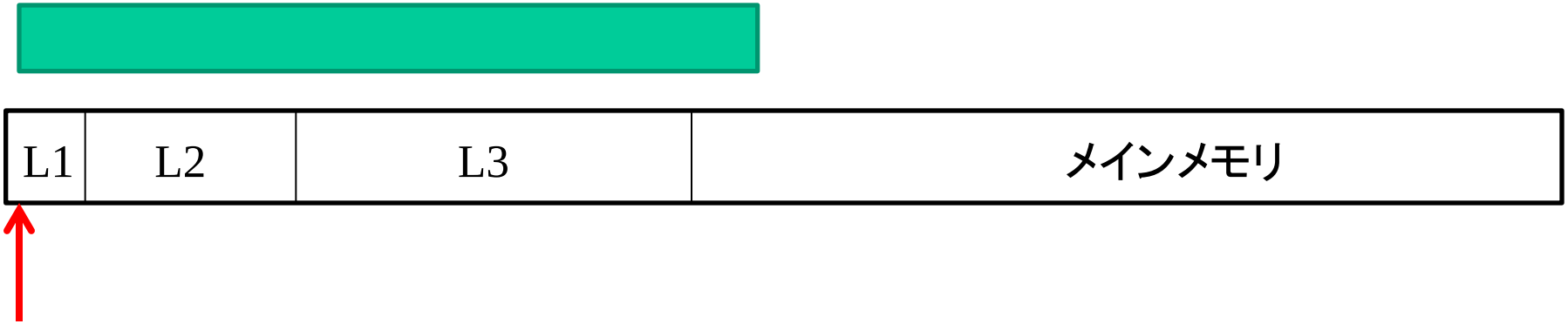
- 階層メモリは下に行くほど低性能



階層メモリへの格納

- スタックが大きくなると上の階層では入らない

スタックの実体：あふれたものが順次下の階層に行く



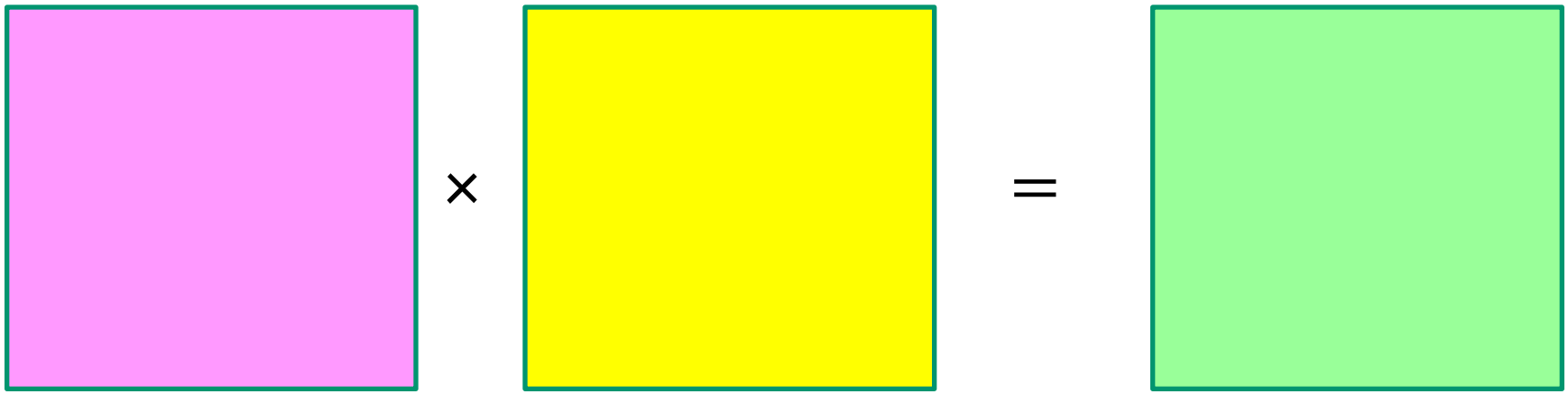
問題2

- 十分に大きい2個の行列の行列積を求め、別の行列として作成するプログラムを作成する。

- 行列サイズを順次大きくし、演算あたりの実行時間変化をリストまたはグラフとして求めよ。

問題1の計算本体部分だけを変更すれば出来る

- これが、どのようにメモリに乗るのか想像しよう
- ループの i,j,kの順を変えたりブロック化して
  振る舞いが変わるか試そう



プログラミング言語

- 速く計算できるならば何でもよい
- 一般に　Fortran ＞ C/C++＞Java＞Ruby

http://www-hiraki.is.s.u-tokyo.ac.jp/lectures/prog_giho/

にプログラム参考例がC, Java, Rubyである。

とりあえず動かし、あとは改造して速くする努力

プログラミング入門1（一般論）

- プログラミング言語は数式ではない
  - 左辺 ⇒ 値を決めるための式
  - 右辺 ⇒ 値を入れる場所の指定

} 場所により意味が違う

  - 例 `r = caesar(in,out);`

    `z = x*y + 4*x*x*y – 1/y;`

    `p = &x;` ポインタの例。ポインタは便利だが毒

- 一度に1個の文しか実行しない
  - 条件分岐が無い限り上から下に順次実行

プログラミング入門2（ヘッダ）

- 言語により色々おまじないが前につく
- Cの場合

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
```

出来合いの関数使う時必要
特に
#include <stdlib.h>
#include <stdio.h>　はいつでもいるおまじない
　これは使わないもの書いても実害は小さい

```
static void usage(void);
static int caesar(FILE *in, FILE *out);
extern int main(int argc, char *argv[]);
```

これは使う関数の返値の型宣言
voidは何も返さない時

これも多すぎる分には無害

その下に順次使う関数を定義（関数は順不同でよい）
そのプログラムを実行するときは、何故かmain()が最初に動く

プログラミング入門3（関数）

- 関数　・・・　関数型言語以外では全く関数でない
  関数の定義を思い起こそう！
  内部に状態（データ等）アクセスがあり同じ入力で違う値

- 引数を入力、関数値を出力とする

  複数の出力出したいときは、関数値でなく
  メモリに値を書いて返す

- 値　⇒　メモリ・レジスタ上の値、関数返値などを演算で結ぶもの

Java入門1

- 前に付けるおまじないはずっと簡単

```
package su.hirakilab;

import java.io.*;

public class Caesar {
        private InputStream m_in;
        private OutputStream m_out;
```

- args[]　mainにコマンドラインから与える値（達）
- ファイルの使い方には癖がある

Java入門2

```
package su.hirakilab;

import java.io.*;

public class Caesar {
        private InputStream m_in;
        private OutputStream m_out;

        public Caesar(InputStream i, OutputStream o) {
                m_in=i;
                m_out=o;
        }

        public void run() throws IOException {
                int c;
                while(0<(c=m_in.read())) {

        if(Character.isLetterOrDigit(c)) { c++; }
                                m_out.write((char)c);
                }
        }
```

Java入門3

```
	public static void main(String[] args) {
		InputStream i=System.in;
		OutputStream o=System.out;
		try {
			if(args.length>0) {
				if(!("-".equals(args[0]))) {
					i=new FileInputStream(args[0]);
				}
			}
			if(args.length>1) {
				o=new FileOutputStream(args[1]);
			}
		} catch (Exception e) {
			System.out.println(e.toString());
			if(i!=null) {
				try { i.close(); } catch (IOException ignored) {}
			}
			System.exit(1);
		}

		Caesar c=new Caesar(i,o);
		try {
			c.run();
		} catch (IOException e) {
			System.err.println(e.toString());
		} finally {
			try { o.close(); } catch (IOException ignored) {}
			try { i.close(); } catch (IOException ignored) {}
		}
	}
}
```

Ruby入門1

- 殆どおまじない無しで簡単

```
#!/usr/bin/ruby

i=STDIN
o=STDOUT

if(ARGV.length>=1) then
        i=File.new(ARGV[0],"r") if("-"!=ARGV[0])
end
o=File.new(ARGV[1],"w") if(ARGV.length>=2)

i.each_char { |c|
        cc=c.unpack("C")[0]
        cc=cc+1 if(/¥w/ =~ c)
        o.putc(cc)
}
o.close
i.close
```

- その代償が遅い事

Ruby入門2

- 僅かにあるおまじない

```
#!/usr/bin/ruby

i=STDIN
o=STDOUT

if(ARGV.length>=1) then
        i=File.new(ARGV[0],"r") if("-"!=ARGV[0])
end
o=File.new(ARGV[1],"w") if(ARGV.length>=2)

i.each_char { |c|
        cc=c.unpack("C")[0]
        cc=cc+1 if(/¥w/ =~ c)
        o.putc(cc)
}

o.close
i.close
```

標準入出力とつなぐ

走らせる時のオプション（引数）

参考文献

- 素人（失礼！）が書いたWebページよりは

Java http://docs.oracle.com/javase/6/docs/api/
Ruby http://docs.ruby-lang.org/ja/2.0.0/doc/
C manページを見る

など、正式の情報を見よう

新しい言語を覚える

- おまじないの形と定型を覚える
- 変数名の使い方を覚える
- あとは実例見ながら少しづつ変えて走らせて試す

C ⇒ ほとんど機械語。どのような命令が出るか見える

Java ⇒ 実際にやっていることは複雑だが一応わかる

Ruby ⇒ 必須の計算以外に沢山命令を使って便利にしている
その結果、どういう命令が実際に走るかはわからない

手続型でない言語

- 関数型言語　　Lisp（Scheme), Haskell, ML, OCaml等多数
  - プログラムの正しさ、安全性の証明可能性に道
- 論理型言語　　Prolog等
  - 双方向言語、制約充足によるプログラミング、並列性に道
- ハードウェア記述言語　　VHDL, Verilog, System Cが主に使われる
  - 本質的に並列記述、並列実行。実行というよりはシミュレーション
- 応用指向言語（Domain Specific Language, DSLも含まれる)
  - R　　統計処理、機械学習に向いた言語
  - MATLAB　　数値解析に向いた言語
  - Z　　形式言語による仕様記述と実行向き
  - SQL　　データベース言語

手続型でない言語

- 関数型言語　　Lisp（Scheme), Haskell, ML, OCaml等多数
  - プログラムの正しさ、安全性の証明可能性に道
- 論理型言語　　Prolog等
  - 双方向言語、制約充足によるプログラミング、並列性に道
- ハードウェア記述言語　　VHDL, Verilog, System Cが主に使われる
  - 本質的に並列記述、並列実行。実行というよりはシミュレーション
- 応用指向言語（Domain Specific Language, DSLも含まれる)
  - R　　統計処理、機械学習に向いた言語
  - MATLAB　　数値解析に向いた言語
  - Z　　形式言語による仕様記述と実行向き
  - SQL　　データベース言語

赤字は理学部情報科学科で学ぶプログラミング言語

チューリング完全の重要性

- チューリングマシン：無限長のテープの上に、チューリングマシンが乗っている。計算可能性の定義に用いられる。

- チューリング完全とは、チューリングマシンが計算できる問題すべてが記述、計算でき、もしメモリが無限にあり、計算時間が無限にかかってよいならば解けることを示す

- チューリング完全でないプログラミング言語では、記述できない計算可能問題が出てしまう。
  - 例えばHTML（Webページの記述言語）はチューリング完全でない

古典的言語

- 言語の良し悪しを論じる必要のない言語たち
- Fortran, Cobol, Basic, VB
  - 現在でも広く使われている
  - 新しいソフト開発というより、既存の熟成したソフトの継続性維持

- 各種スクリプト言語　もともとは非常に遅かった
  - Shell script, AWK　古典的、UNIXのおまけ
  - Perl, Python, PHP, Javascript, Ruby　　最近は普通の言語なみ？の速度

- アセンブリ言語　　要するに読める機械語
  - C言語のおかげで日陰者になった
  - リアルタイムの世界では現在でも多用される