

# プログラミング応用

Week9

# 前期期末試験までの内容

- アルゴリズムの話が中心
  - 再帰アルゴリズム(9週、10週)
  - ソートアルゴリズム(11週、12週)
  - 探索アルゴリズム(13週、14週)

大学編入、大学院入試の情報系試験では  
ほぼ間違いなく出題される分野  
4年生選択科目にも「データ構造とアルゴリズム」がある  
3年生では基本的なところのみ学習

# 自習用参考書



柴田望洋、辻亮介  
「C言語によるアルゴリズムと  
データ構造」

# 本日の講義

- アルゴリズムとは？
  - アルゴリズムの定義
  - 良いアルゴリズム
- 再帰アルゴリズム
  - 再帰処理の復習
    - 階乗計算アルゴリズム、フィボナッチ数列
- 演習
  - timeコマンドを使ってみる
  - 再帰処理の復習

# アルゴリズムの定義

問題を解くためのものであって、明確に定義され、  
順序付けられた有限個の規則からなる集合  
(JIS規格による定義)

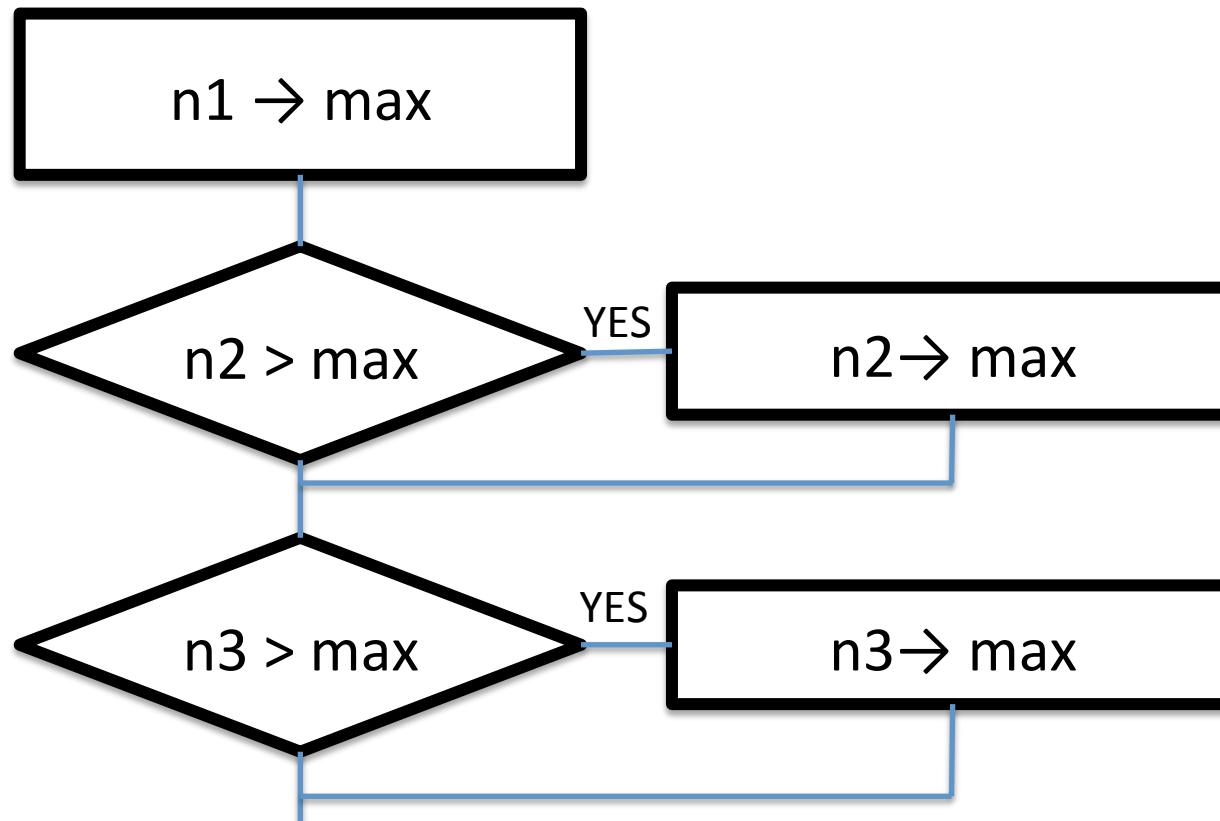
# 三値の最大値を求めるプログラム

- 以下のプログラムを例に  
「アルゴリズム」とはなにか？を考える

```
1 #include <stdio.h>
2
3 int main(void) {
4     int n1, n2, n3, max;
5     printf("3つの値を入力してください。 \n");
6     // scanf()関数で3つの整数をキーボードから読み込む
7     scanf("%d", &n1);
8     scanf("%d", &n2);
9     scanf("%d", &n3);
10    // n1, n2, n3の最大値を求める
11    max = n1;
12    if (n2 > max) max = n2;
13    if (n3 > max) max = n3;
14    // 結果表示
15    printf("最大値は%dです。 \n", max);
16 }
```

# フローチャートによる表現

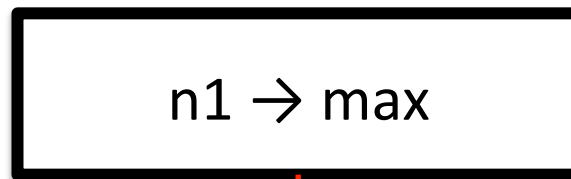
```
10 // n1, n2, n3の最大値を求める
11 max = n1;
12 if (n2 > max) max = n2;
13 if (n3 > max) max = n3;
```



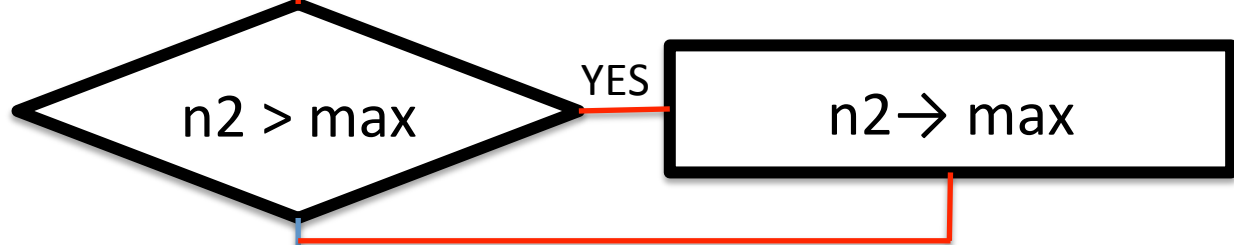
# 変数maxの値変化

- $n1 = 1, n2 = 2, n3 = 3$  のとき

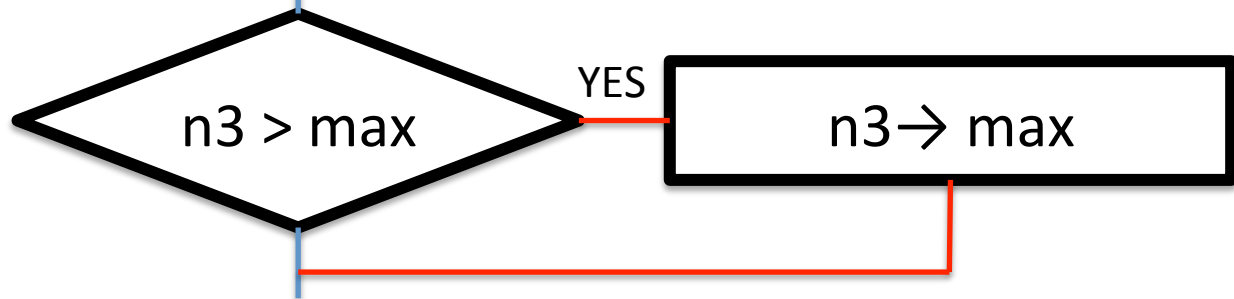
max: 1



max: 2



max: 3

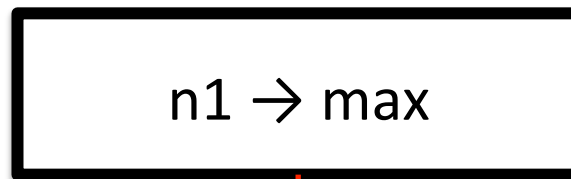




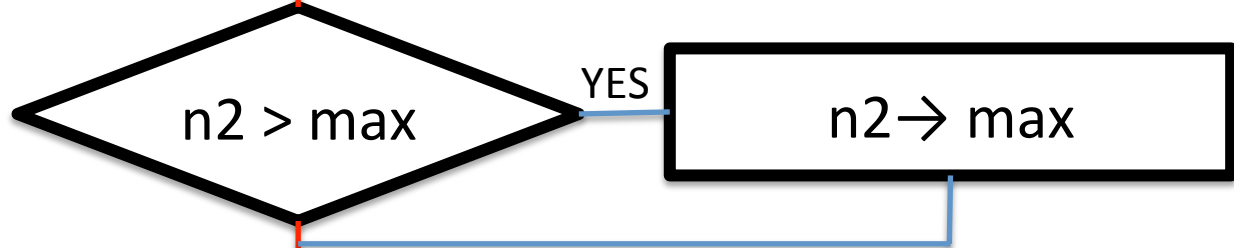
# 変数maxの値変化

- $n1 = 2, n2 = 1, n3 = 3$  のとき

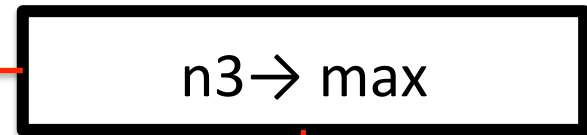
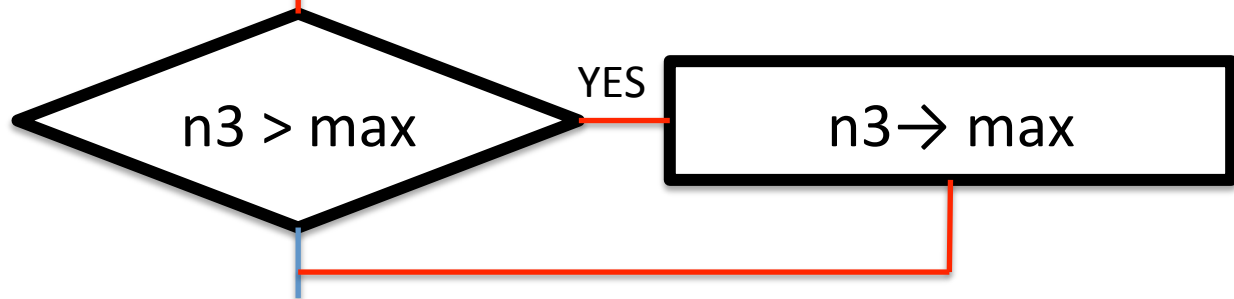
max: 2



max: 2



max: 3



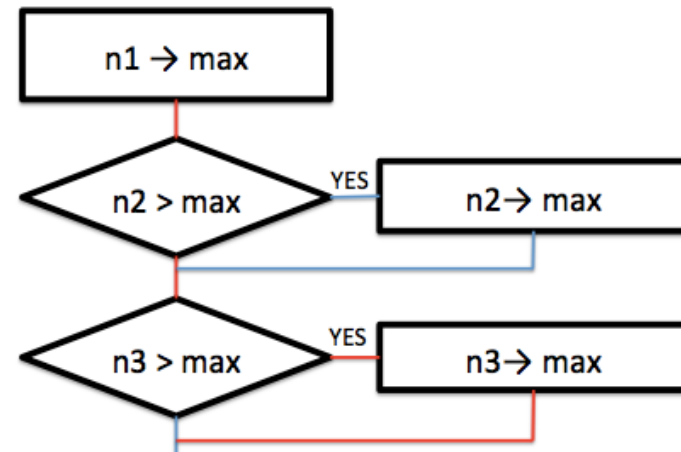
# アルゴリズムの定義(再)

問題を解くためのものであって、明確に定義され、  
順序付けられた有限個の規則からなる集合

# アルゴリズムの定義(再)

問題を解くためのものであって、明確に定義され、  
順序付けられた有限個の規則からなる集合

三値の最大値を求める  
という問題



# 三値の最大値アルゴリズム(2)

- 三値の最大値を求めるアルゴリズムは他にもあるだろうか？
  - アイデア:すべての条件を列挙してif文で記述

n1が最大値である条件

$$n1 > n2 > n3$$

$$n1 > n2 = n3$$

$$n1 > n3 > n2$$

$$n1 = n3 > n2$$

$$n1 = n2 > n3$$

$$n1 = n2 = n3$$

n2が最大値である条件

$$n2 > n1 > n3$$

$$n2 > n1 = n3$$

$$n2 > n3 > n1$$

$$n2 = n3 > n1$$

# 良いアルゴリズムとは何か(1)

```
10 // n1, n2, n3の最大値を求める
11 max = n1;
12 if (n2 > max) max = n2;
13 if (n3 > max) max = n3;
```

2つのアルゴリズム  
どちらが**良い**?

---

```
10 // n1, n2, n3の最大値を求める(すべての条件
    列挙)
11 if ((n1 > n2 && n2 > n3) || (n1 > n2 && n2
    == n3) || (n1 > n3 && n3 > n2) || (n1 == n3
    && n3 > n2) || (n1 == n2 && n2 > n3) || (n1
    == n2 && n2 == n3)) {
12     max = n1;
13 } else if ((n2 > n1 && n1 > n3) || (n2 > n
    1 && n1 == n3) || (n2 > n3 && n3 > n1) || (n
    2 == n3 && n3 > n1)) {
14     max = n2;
15 } else {
16 }
```

# 良いアルゴリズム

## 1. 計算量が少ない

- 比較、計算、代入などの処理がより少なく済むアルゴリズムは計算量が少ない(=速い)
- このところの情報分野では扱うデータが大容量化。高速に処理できるアルゴリズムが求められる。

## 2. メモリ使用量が少ない

- 物理メモリ量を使い果たすと処理が急激に遅くなる(スワップ)

## 3. (実用的にはより簡潔に記述できたほうが良い)

# アルゴリズムの定義(再)

問題を解くためのものであって、明確に定義され、  
順序付けられた有限個の規則からなる集合

解く問題としては  
以下がよく扱われる

1. 探索
2. 並べ替え

\* 第11週以降に学ぶ

より簡潔に、効率的に記述するための手法として  
「再帰」が用いられることが多い

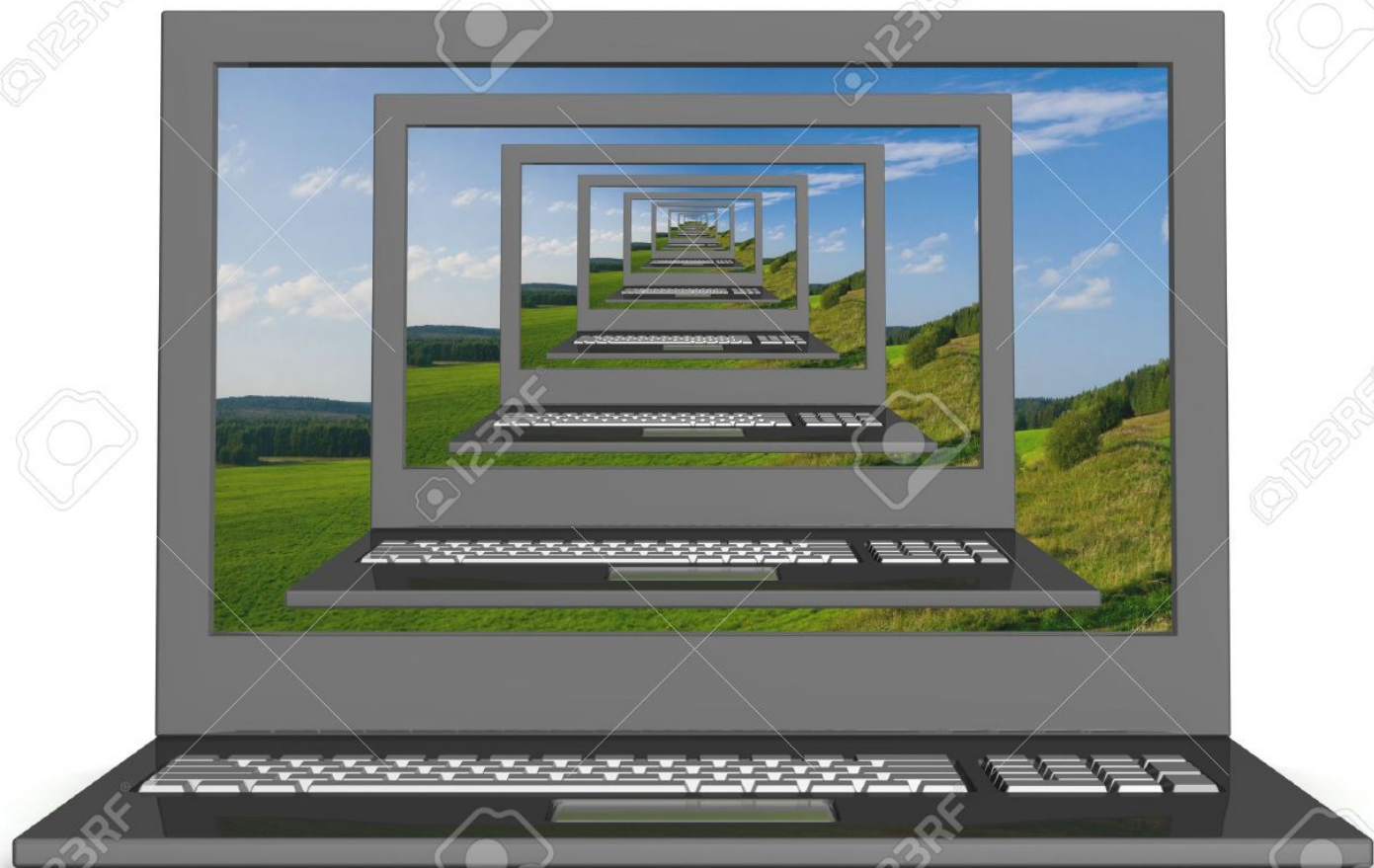
# 本日の講義

- アルゴリズムとは？
  - アルゴリズムの定義
  - 良いアルゴリズム
- 再帰アルゴリズム
  - 再帰処理の復習(階乗計算アルゴリズムの例)
- 演習
  - timeコマンドを使ってみる
  - 再帰処理の復習



# 再帰

- 再帰的(recursive)
  - ある事象が自分自身を含んでいたたり、それを用いて定義されている状態
- 再帰アルゴリズム(Recursive Algorithm)
  - 再帰を用いより簡潔に書かれたアルゴリズム
  - 再帰を用いない場合よりもしばしば効率的



# 階乗計算の例

- 階乗を計算する再帰アルゴリズム

$$5! = 5 * 4 * 3 * 2 * 1$$

```
1 #include <stdio.h>
2
3 int recursive_kaijou(int number) {
4     if (number > 0 ) {
5         return number * recursive_kaijou(number -1);
6     } else {
7         return 1;
8     }
9 }
10
11 int main(void) {
12     printf("5の階乗は%d\n", recursive_kaijou(5));
13 }
```

# 階乗計算の動作(1)

1. main()関数からrecursive\_kaijou(5)が呼び出される
2. 5 \* recursive\_kaijou(4)が実行される
3. 4 \* recursive\_kaijou(3)が実行される
4. 3 \* recursive(2)が実行される
5. 2 \* recursive(1)が実行される

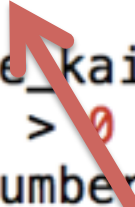
```
3 int recursive_kaijou(int number) {  
4     if (number > 0 ) {  
5         return number * recursive_kaijou(number -1);  
6     } else {  
7         return 1;  
8     }
```

# 階乗計算の動作(1)

1. main()関数からrecursive\_kaijou(5)が呼び出される
2. 5 \* recursive\_kaijou(4)が実行される
3. 4 \* recursive\_kaijou(3)が実行される
4. 3 \* recursive(2)が実行される
5. 2 \* recursive(1)が実行される

```
3 int recursive_kaijou(int number) {  
4     if (number > 0) {  
5         return number * recursive_kaijou(number - 1);  
6     } else  
7         return 1;  
8 }
```

recursive(1)は1を返す



# 階乗計算の動作(1)

1. main()関数からrecursive\_kaijou(5)が呼び出される
2. 5 \* recursive\_kaijou(4)が実行される
3. 4 \* recursive\_kaijou(3)が実行される
4. 3 \* recursive(2)が実行される
5. 2 \* recursive(1)が実行される

```
3 int recursive_kaijou(int number) {  
4     if (number > 0) {  
5         return number * recursive_kaijou(number - 1);  
6     } else  
7         return 1;  
8 }
```

recursive(2)は2を返す

recursive(1)は1を返す

# 階乗計算の動作(1)

1. main()関数からrecursive\_kaijou(5)が呼び出される
  2. 5 \* recursive\_kaijou(4)が実行される
  3. 4 \* recursive\_kaijou(3)が実行される
  4. 3 \* recursive(2)が実行される
  5. 2 \* recursive(1)が実行される
- recursive(3)は6を返す

```
3 int recursive_kaijou(int number) {
4     if (number > 0) {
5         return number * recursive_kaijou(number - 1);
6     } else {
7         return 1;
8     }
}
```

recursive(2)は2を返す

recursive(1)は1を返す

# 階乗計算の動作(1)

1. main()関数からrecursive\_kaijou(5)が呼び出される
2. 5 \* recursive\_kaijou(4)が実行される
3. 4 \* recursive\_kaijou(3)が実行される recursive(4)は24を返す
4. 3 \* recursive(2)が実行される recursive(3)は6を返す
5. 2 \* recursive(1)が実行される recursive(2)は2を返す

```
3 int recursive_kaijou(int number) {
4     if (number > 0) {
5         return number * recursive_kaijou(number - 1);
6     } else {
7         return 1;
8     }
}
```

recursive(1)は1を返す



# 階乗計算の動作(1)

1. main()関数からrecursive\_kaijou(5)が呼び出される
2. 5 \* recursive\_kaijou(4)が実行される
3. 4 \* recursive\_kaijou(3)が実行される
4. 3 \* recursive(2)が実行される
5. 2 \* recursive(1)が実行される

recursive(5)は24\*5を返す

recursive(4)は24を返す

recursive(3)は6を返す

recursive(2)は2を返す

recursive(1)は1を返す

```
3 int recursive_kaijou(int number) {
4     if (number > 0) {
5         return number * recursive_kaijou(number - 1);
6     } else {
7         return 1;
8     }
}
```

# 演習0

- GitHubのシラバスページから以下の2つのファイルをダウンロード
  - 三値の最大値プログラム  
week9/maximum3.c
  - 三値の最大値プログラム(すべての条件列挙)  
week9/maximum3\_slow.c

# 演習1

- ダウンロードしたファイルを以下のコマンドで名前付きコンパイル  
(-oオプションを付けると、実行ファイルがa.outではなく指定した名前で作成される)
  - \$ cc maximum3.c -o maximum3
  - \$ cc maximum3\_slow.c -o maximum3\_slow

# 演習2

- 以下のシェルスクリプトを10000times.shという名前で保存し実行出来ることを確認
  - 補足1: `seq 1 10000`は1から10000までの数列を列挙するUNIXコマンド
  - 補足2: このシェルスクリプトは./maximum3の実行を10000回繰り返す

```
1 #!/bin/bash
2 for i in `seq 1 10000`
3 do
4     echo "${i}回目の実行";
5     ./maximum3
6 done
```

# 演習3

- 以下のシェルスクリプトを10000times\_slow.shという名前で保存し実行できることを確認
  - 補足: このシェルスクリプトはmaximum3\_slowを10000回実行するプログラムになっている

```
1 #!/bin/bash
2 for i in `seq 1 10000`
3 do
4     echo "${i}回目の実行";
5     ./maximum3_slow
6 done
```

# 演習4

- 以下のコマンドで10000times.shと10000times\_slow.shの処理が完了するまでの時間を計る
  - \$ time ./10000times.sh
  - \$ time ./10000times\_slow.sh
- 補足：  
10000回の実行では実行時間は変わらない。  
(この演習はtimeコマンドの使い方を学ぶものと割りきって欲しい)  
第11週以降に学ぶソートや探索では、アルゴリズムによって計算時間に大きな差が出る。

# 演習5

- 以下のプログラム(kaijou.c)を作成しなさい
    - 再帰を用いて階乗を計算する(kaijou\_recursive.c)
      - 階乗を計算した値を返す、以下の形式の関数を作成  
`int recursive_kaijou(int number);`
    - 再帰を用いずに階乗を計算する(kaijou.c)
      - 階乗を計算した値を返す、以下の形式の関数を作成  
`int kaijou(int number);`
- \* いずれのプログラムもmain()から作成した関数を呼び出し正しく動作することを確認すること。
- \* 階乗を用いることでより簡潔にプログラムが記述できることを確認すること。

# 演習6

- フィボナッチ数を計算するプログラム(fibo\_recursive.c)を再帰を用いて作成しなさい
  - なおプログラム内の関数の形式は以下とする  
`int fibo_recursive(int i);`
- 自然数 $n$ のフィボナッチ数は以下の条件を満たす
  - $i = 0$  の時  $\text{fibo}(0) = 0$
  - $i = 1$  の時  $\text{fibo}(1) = 1$
  - $i = 2$  の時  $\text{fibo}(2) = 1$
  - $i = 3$  の時  $\text{fibo}(3) = 2$
  - $i = 4$  の時  $\text{fibo}(4) = 3$
  - $i = 5$  の時  $\text{fibo}(5) = 5$
  - $i = n$  の時  $\text{fibo}(n) = \text{fibo}(n - 1) + \text{fibo}(n - 2)$



# 演習7

- フィボナッチ数を計算するプログラム(fibo.c)を再帰を用いずに作成しなさい
  - なお形式は以下とする  
`int fibo(int x);`
- 自然数 $n$ のフィボナッチ数は以下の条件を満たす
  - $i = 0$  の時  $\text{fibo}(0) = 0$
  - $i = 1$  の時  $\text{fibo}(1) = 1$
  - $i = 2$  の時  $\text{fibo}(2) = 1$
  - $i = 3$  の時  $\text{fibo}(3) = 2$
  - $i = 4$  の時  $\text{fibo}(4) = 3$
  - $i = 5$  の時  $\text{fibo}(5) = 5$
  - $i = n$  の時  $\text{fibo}(n) = \text{fibo}(n - 1) + \text{fibo}(n - 2)$

# 次週

- 再帰アルゴリズムをさらに深掘りします。
  - 複雑な再帰アルゴリズムの動作を解析する方法
  - 有名な再帰アルゴリズム(最大公約数、ハノイの塔)
- 課題が終わっていない人は放課後等に演習室に来て進めておきましょう。