プログラミング応用演習 I
Applied Programming 1

1st, 2nd Class

C言語の使い方の振り返り、
関数再帰呼び出し

計算機環境の使い方

- ログイン / login
  - https://imc.tut.ac.jp/classroom/guide.html#chapter16
    - Windowsからリモートデスクトップ xdev.edu.tut.ac.jp
  - Linux環境の利用方法

- マルチメディア教室以外で課題を行いたいときは
  - 学内なら
    - 学内ネットワークにつないでリモートデスクトップをつなぐ
  - 学外なら
    - VPNを使う（学内ネットワークと同じになる）
    - https://imc.tut.ac.jp/network/vpn
    - VPNを使っている状態でリモートデスクトップをつなぐ

プログラムの書き方

- テキストエディタ -> ソースコードを編集するためのもの
  - Visual Studio Code (Windows, Mac)
    - C言語用の拡張を入れる
    - sshできる拡張を入れる
      - https://github.com/IRSL-tut/cps_rpi_docker?tab=readme-ov-file#vscode%E3%82%92%E4%BD%BF%E3%81%A3%E3%81%9Fraspberry-pi%E3%81%B8%E3%81%AE%E6%8E%A5%E7%B6%9A
  - Emacs (Linux)
  - Vim (Linux)
- ファイルを転送する
  - WinSCP https://winscp.net/eng/index.php
- ターミナルで接続する(ssh)
  - TeraTerm https://teratermproject.github.io/

プログラムの実行方法

- ソースコードを書く
  - リモートデスクトップ先もしくは手元で書く
  - 手元で書いたらテキストファイルを計算機環境へ入れる
- ソースコードをコンパイル
  - ターミナル上で実行
- コンパイルしてできたバイナリを実行
  - ターミナル上で実行
- 実行結果を保存
  - 手元にコピーペースト

- ターミナルの使い方
  - https://imc.tut.ac.jp/classroom/guide.html#chapter22

C言語基本の振り返り / プログラムの構成

- C言語ソースコードの最小構成

```c
/* header files */
#include <stdio.h>

/* functions */
int function_name(int n); // 関数宣言

/* main 関数 */
int main(void) {
/*
  プログラム記述 main
*/
  return 0;
}

int function_name(int n)
{
/*
  プログラム記述 function
*/
}
```

/* と */ で囲われた範囲はコメントとして無視されます。
（プログラムに影響しません）

// と書いた行末までも同じくコメントです。

プログラムの実行方法

- ソースコードを書く（前頁のコードをコピーペーストする）
  - リモートデスクトップ先もしくは手元で書く
  - 手元で書いたらテキストファイルを計算機環境へ入れる
  - hello.cという名前にする
- ソースコードをコンパイル
  - ターミナル上で実行
  - $ gcc hello.c
- コンパイルしてできたバイナリを実行
  - ターミナル上で実行
  - $ ./a.out
- 実行結果を保存
  - 手元にコピーペースト

$ xxx yyy zzz
と書かれていたらターミナルでの入力を示します。
先頭の$を無視してそれ以降をターミナルへ入力します。

ターミナルは以下のような入力待ちが出ることが多いため
$始まりで示しています。
username@machinename:~/directory$

$ gcc hello.c -o hello ## -o hello をつけるとhelloという名前で出力される
$ ./hello ## ./a.outと同じく実行できる

C言語基本の振り返り / プログラムの構成

- C言語ソースコードの最小構成

```c
/* header files */
#include <stdio.h>

/* functions */
int function_name(int n); // 関数宣言

/* main 関数 */
int main(void) {
/*
  プログラム記述 main
*/
  return 0;
}

int function_name(int n) // 関数の実体
{
/*
  プログラム記述 function
*/
  return 1;
}
```

#include <xxx> はヘッダーファイルのインクルードです
（関数を使えるようにします）

関数宣言した後の記述で関数が使えます。

main関数が、コンパイルして実行したときに実行されます

C言語基本の振り返り / 変数と演算1

- 変数型
  - 整数 int（その他、char、short、long等あります、長さが違います）
  - 浮動小数点 double（そのほか float等あります）
- 演算子
  - + - * 足し算、引き算、掛け算
  - / (a / b) 割り算a, bが整数のみの場合と、浮動小数点がある場合に結果が異なる
  - % (a % b) 剰余(余り) a と b は整数である必要
- 代入
  - a = 1; //aに1を代入
- 比較
  - == != > >= < <=
  - (a == b) //真偽値として扱われる

C言語基本の振り返り / 変数と演算2

- 変数宣言
  - `int a;`
- 配列
  - `int a[5];`
- 構造体（3週目に説明があります）
  - `struct sname { /*structの内容*/ };`
- ポインター（3週目に少し説明があります）
  - アドレスを扱う（発展的内容です）
- 文字列
  - char型(整数)の配列として表す
  - 文字数 + 1個の配列を確保して、配列の最後は `'¥0'` を入れる

C言語基本の振り返り / 入出力1

- 出力 puts関数 ターミナルへの文字と改行の出力
  - 文法 puts(文字列);
  - 例 puts("This is a sample");
- 出力 printf関数 ターミナルへの文字出力(フォーマット版)
  - 文法 printf(フォーマット文字列, 変数, …);
  - 例 printf("a = %d, b = %f, c = %s¥n", a, b, c);
  - %d は整数型の表示　%f は浮動小数点型の表示
  - %s は文字列の表示　¥n は改行文字列
  - フォーマット文字列の % が現れる順番に 変数をカンマ区切りで記述
- 出力 fputs, fprintf関数 (出力先の指定)
  - 文法 fprintf(FILE構造体, フォーマット文字列, 変数, …);
  - 文法 fputs(FILE構造体, 文字列, 変数);u
  - fprintf(stdout, …); は printfと同じ
  - fprintf(stderr, …); はターミナルのエラー出力への文字出力

C言語基本の振り返り / 入出力2

- 関数の
  - scanf関数
  - scanf(フォーマット文字列, &var);

scanfの例

```
int ivar; // 値を入れる変数
scanf("%d", &ivar);

double fvar; // 値を入れる変数
scanf("%f", &fvar);
```

  - fget関数
  - fget(文字列変数, 読み込みサイズ, 読み込む相手);

fgetの例 / 整数型

```
int ival; // 値を入れる変数
char str[256]; // 一時的に使う文字列
fgets(str, sizeof(str), stdin);
ival = atoi(str);
```

fgetの例 / 浮動小数点型

```
double fval; // 値を入れる変数
char str[256]; // 一時的に使う文字列
fgets(str, sizeof(str), stdin);
fval = atof(str);
```

C言語基本の振り返り / 関数の呼び出し

- 関数の呼び出し

```c
/* header files */
#include <stdio.h>

/* functions */
int function_name(int n, int m); // 関数宣言

/* main 関数 */
int main(void) {
  int result;
  int a = 10;
  int b = 20;
  result = function_name(a, b);
  return 0;
}

int function_name(int n, int m) // 関数の実体
{
/*
  プログラム記述 function
*/
  return n + m;
}
```

関数宣言した後の記述で関数が使えます。

関数宣言した function_name が関数名です。

先頭のint は返り値の型です。
（なにも返さない場合は void ）

() カッコ内の int n, int m は引数 です。

C言語基本の振り返り / 条件分岐、繰り返し

- 条件分岐 if 文

```
if ( 条件式 ) {
/* 文 */
}
```

条件式が 真 (値としては 0 以外) の時に**文**が実行される。

条件式は値が返るものである。

条件式の例: a > 1 など

```
if ( 条件式1 ) {
/* 文1 */
} else if ( 条件式2 ) {
/* 文2 */
} else {
/* 文3 */
}
```

条件式1が 真 (値としては 0 以外) の時に**文1**が実行される。

条件式1が 真 (値としては 0 以外) でない場合で、**条件式2**が 真 (値としては 0 以外) の時に**文2**が実行される。

else if は何個でも続けることができます

条件式1, 条件式2 が、いずれも真でないとき、**文3**が実行される。

C言語基本の振り返り / 条件分岐、繰り返し

- 繰返し for文

```
for(初期化 ; 条件式 ; 変化式 ) {
/* 文 */
}
```

例: 文が10回実行される

```
for(i = 0; i < 10; i++ ) {
/* 文 */
}
```

1. 最初に、**初期化**が行われる。

2. **条件式**が 偽 (値としては 0) の時に実行が終了する。

3. **文** が実行される。

4. **変化式** が実行され、2.に戻る。

- 繰返し while文

```
while( 条件式 ) {
/* 文 */
}
```

1. **条件式**が 偽 (値としては 0) の時に実行が終了する。

2. **文** が実行される。

3. 1.に戻って繰り返す。

ターミナルの使い方（参考）

- https://imc.tut.ac.jp/classroom/guide.html#chapter22
- 分からないことはTAに聞くか、linux コマンド　などで検索してください
- ls / ls −l / ls −tarl
  - ディレクトリ(windowsでのフォルダ)の中身を表示
- cd
  - ディレクトリの移動
  - .. は一つ上のディレクトリ
  - . は現在のディレクトリ
  - ディレクトリの階層は / で表す
- mkdir
  - ディレクトリの作成
- rm / rm −r
  - rm −r はディレクトリを消す場合（その下のファイルも消える）
- man
  - マニュアルを見る

振り返りのまとめ

- ここで、sample_0.c をコンパイルして実行してみましょう。

- Google classroomの第1回資料から sample_0.c をダウンロード
- Linux環境に入れて
- コンパイル（実行ファイルができる）
- 実行ファイルを実行
- 実行結果をテキストで手元PCへコピー

第1回の課題の説明

Description of 1st assignment

関数の再帰呼び出し(Recursive function calls)

- ある関数の中で自身と同じ関数を呼ぶ
  - 記述がシンプルになる
  - 繰返しと同じ効果が得られる場合もある
    - 問題の性質に依存する
  - 無限繰り返しとならないように注意して設計する
    - 終了条件を適切にする
      - ある条件で再帰呼び出しを行わないで関数を終了する
      - 終了する条件に到達する
  - 関数の呼び出し回数が多くなりすぎないように注意する

関数の再帰呼び出し（Recursive function calls）

```c
/* header files */
#include <stdio.h>

/* functions */
int fact_loop(int n); // 関数宣言
int fact_recf(int n); // 関数宣言

/* main */
int main(void) {
  int n;
  printf("The factorial of a positive integer n: ");
  scanf("%d", &n);
  printf("LOOP %d! = %d¥n", n, fact_loop(n));
  printf("RECF %d! = %d¥n", n, fact_recf(n));
  return 0;
}
```

ループ版のfact_loop関数
再帰版のfact_recf関数

```c
/* factorial calculation
   using a for-statement */
int fact_loop(int n) {
 int i, result = 1;
  for(i = n; i >= 1; i--) {
    result *= i;
  }
  return result;
}

/* factorial calculation
   using a recursive function call */
int fact_recf(int n) {
  if (n == 1) {
    return 1;
  } else {
    return n * fact_recf(n - 1);
  }
}
```

関数の再帰呼び出し (Recursive function calls)

- ここでは、階乗計算をループを使った場合と、再帰呼び出しを使った場合の二通りの関数を示しています。
- 左がmain関数になります。ループ版の関数fact_loopと再帰版の関数fact_recfの二つの関数を、同じ引数nで呼び出しています。
- 右上の関数がループ版のfact_loop関数です。For文を使って、ループでの実装となります。Forループの変数iについて、渡された引数nから始めて、1ずつ引きながら積を計算していることがわかります。iが1より大きい間、この積が計算されるので nから1までの数値を全部掛け合わせることになり、すなわち階乗の計算をおこなっていることがわかります。
- 一方、右下の関数が再帰版のfact_recf関数です。まず，最初にnが1かどうかを判定し、nが1の場合に、1を返します。この条件文を終了条件と呼び、再帰関数を作る場合には重要となりますので注意してください。そして、nが1以外の場合は、渡されたnとfact_recf(n-1)を掛けたものを返しています。このように、関数内に自分自身を呼び出すことを再帰(recursion)と呼びます。

関数の再帰呼び出し（Recursive function calls）

```c
/* main */
int main(void) {
  int n;
  printf("The factorial of a positive integer n: ");
  scanf("%d", &n);
  …
  printf("RECF %d! = %d¥n", n, fact_recf(n));
  return 0;
}
```

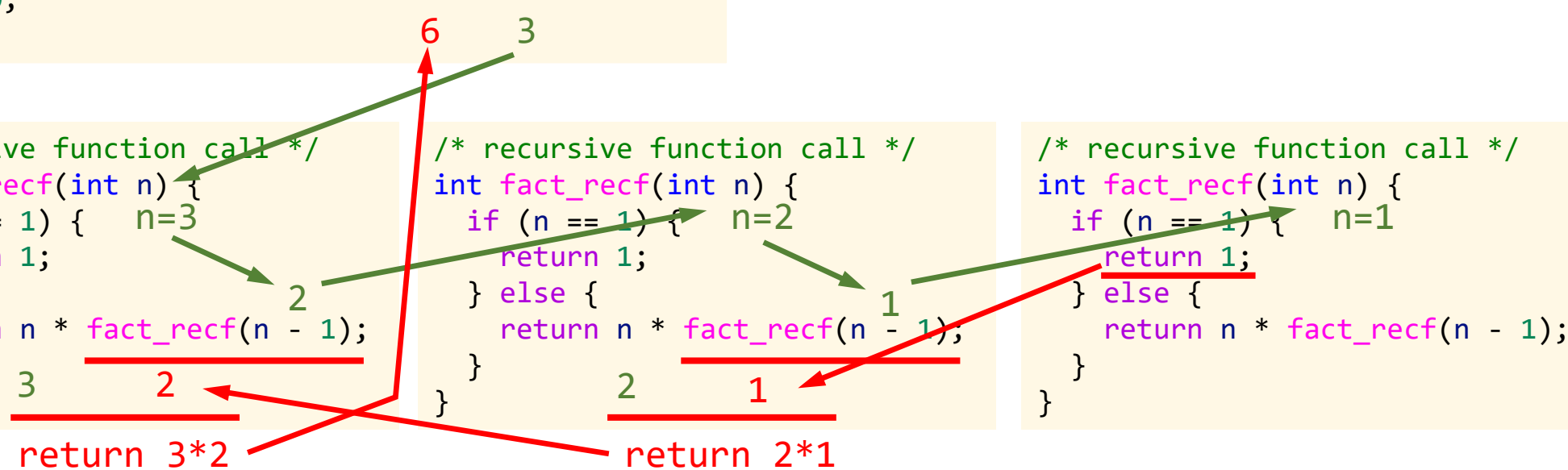
実行結果

The factorial of a positive integer n: 3 //3を入力
RECF 3! = 6

```c
/* recursive function call */
int fact_recf(int n) {
  if (n == 1) {
    return 1;
  } else {
    return n * fact_recf(n - 1);
  }
}
```

```c
/* recursive function call */
int fact_recf(int n) {
  if (n == 1) {
    return 1;
  } else {
    return n * fact_recf(n - 1);
  }
}
```

```c
/* recursive function call */
int fact_recf(int n) {
  if (n == 1) {
    return 1;
  } else {
    return n * fact_recf(n - 1);
  }
}
```



関数の再帰呼び出し (Recursive function calls)

- 再帰版の関数fact_recf関数の実際の動作を見ていきます。
- まず、左上のmain関数において n=3 が入力されたとします。そして、その引数の 3 が fact_recf に渡された場合が左下の図になります。ここでは 3 が渡されていますので終了条件は満足されず、else以降が実行され、return 部分の n*fact_recf(n-1) つまり 3*fact_recf(2) を計算します。
- そして、このfact_recf(2)を計算するために、自分自身(と同じ関数)を新しい引数2で呼び出します。引数2で呼び出されたfact_recf関数が中央下の図です。ここでは引数に 2 が渡されており、先程と同様に、終了条件は満足されずelse以降が実行され、return 部分の n*fact_recf(n-1)、つまり 2*fact_recf(1) を計算します。ここで、自分自身(と同じ関数)を引数1で呼び出します。

アルゴリズムの説明1

- 階乗計算(Factorial calculation)
  - N! = N ＊ N-1 ＊ N-2 ＊ … ＊ 2 ＊ 1 （ループ表現）
  - N! = N ＊ (N-1)! (再帰表現)
- ユーグリッドの互除法 / 2つの数の最大公約数を求める
  - 1. 2つの正の整数を m, n (m≧n)とする。
  - 2. m を n で割った余りが 0 なら、n を最大公約数として終了する。
  - 3. m を n で割った余りを n に代入し、さらに元の n を m に代入して 2 に戻る。
- 素数判定
  - 素数とは「1とその数自身との外には約数がない正の整数」
  - 簡単には与えられた数をNとして、2からN-1まで余りを求めて、余り0が無ければ素数。
  - 試し割り法 / 実際は√N 以下の数を確かめれば良い

第2回の課題の説明

Description of 2nd assignment

アルゴリズムの説明2（2回目以降の課題）

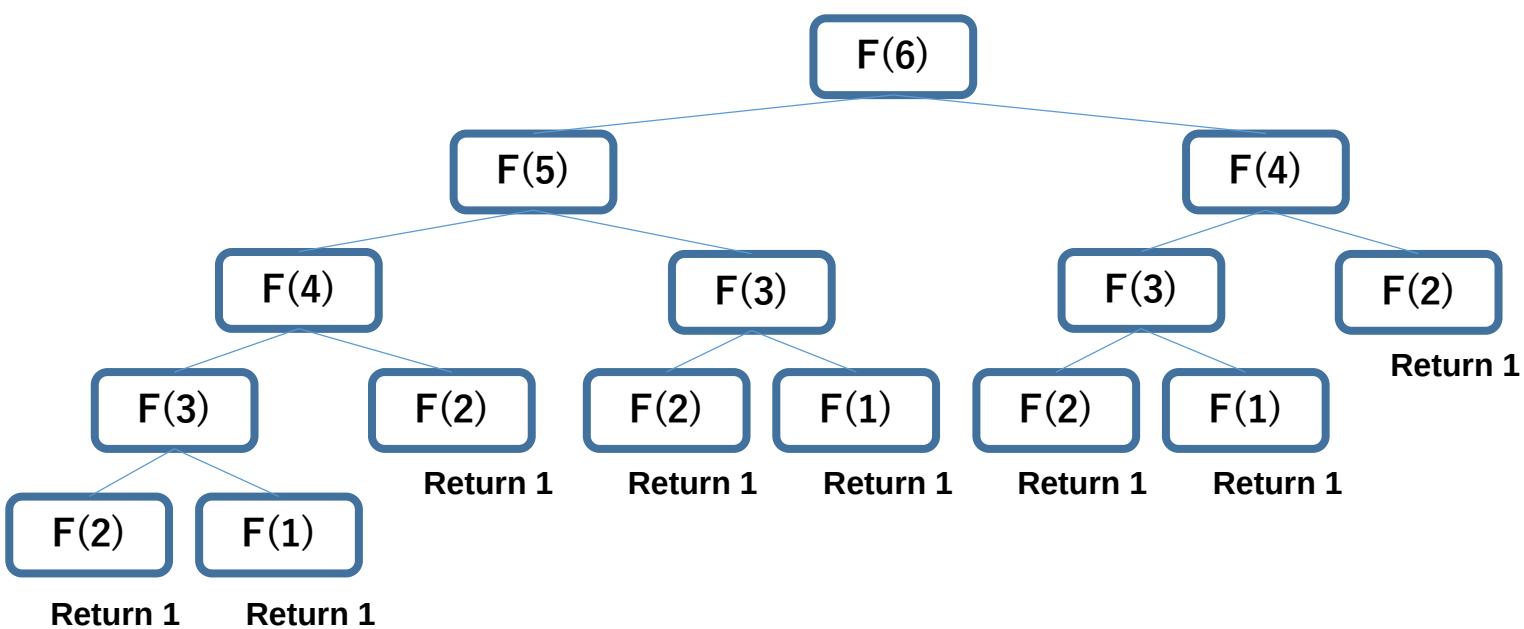
- フィボナッチ数列(Fibonacci number)
  - 自身のコピーを一つ作ることのできる（生）物がいます。
  - 2か月後から毎月自身のコピーを作ります。
  - 死んだり、いなくなったりしません。
  - Nヶ月後には、合計でいくつになっているかを示すのがフィボナッチ数です。
- 素因数分解
  - 素数が関係する
  - メモ化再帰を使えるか
- べき乗計算(Exponentiation, Power)
  - $A^X = A * A * A * \cdots * A$
  - $A^X = A * A^{X-1}$
  - 単純にすると再帰が深くなる

アルゴリズムの説明2（2回目以降の課題）

- フィボナッチ数列(Fibonacci series)
  - There are (live) things that can make one copy of themselves.
  - After two months, it will make copies of itself every month.
  - They do not die or disappear.
  - After N months, the Fibonacci number indicates how many in total.
- 素因数分解 (Prime factor decomposition)
  - Express the number in the form of a product of prime numbers
  - Does it solve by using ″memoized recursion“?
- べき乗計算 (Exponentiation, Power)
  - $A^X = A * A * A * \cdots * A$
  - $A^X = A * A^{X-1}$
  - Simple implementation leads to deep recursion

フィボナッチ数列(Fibonacci number)

- フィボナッチ数列(Fibonacci number)
  - 自身のコピーを一つ作ることのできる(生)物が1ついます。
  - 2か月後から毎月自身のコピーを作ります。
  - 死んだり、いなくなったりしません。
  - Nヶ月後には、合計でいくつになっているかを示すのがフィボナッチ数です。
- 数式で表すと F(N) = F(N−1) + F(N−2) となります。このとき、F(1) = 1, F(2) = 1

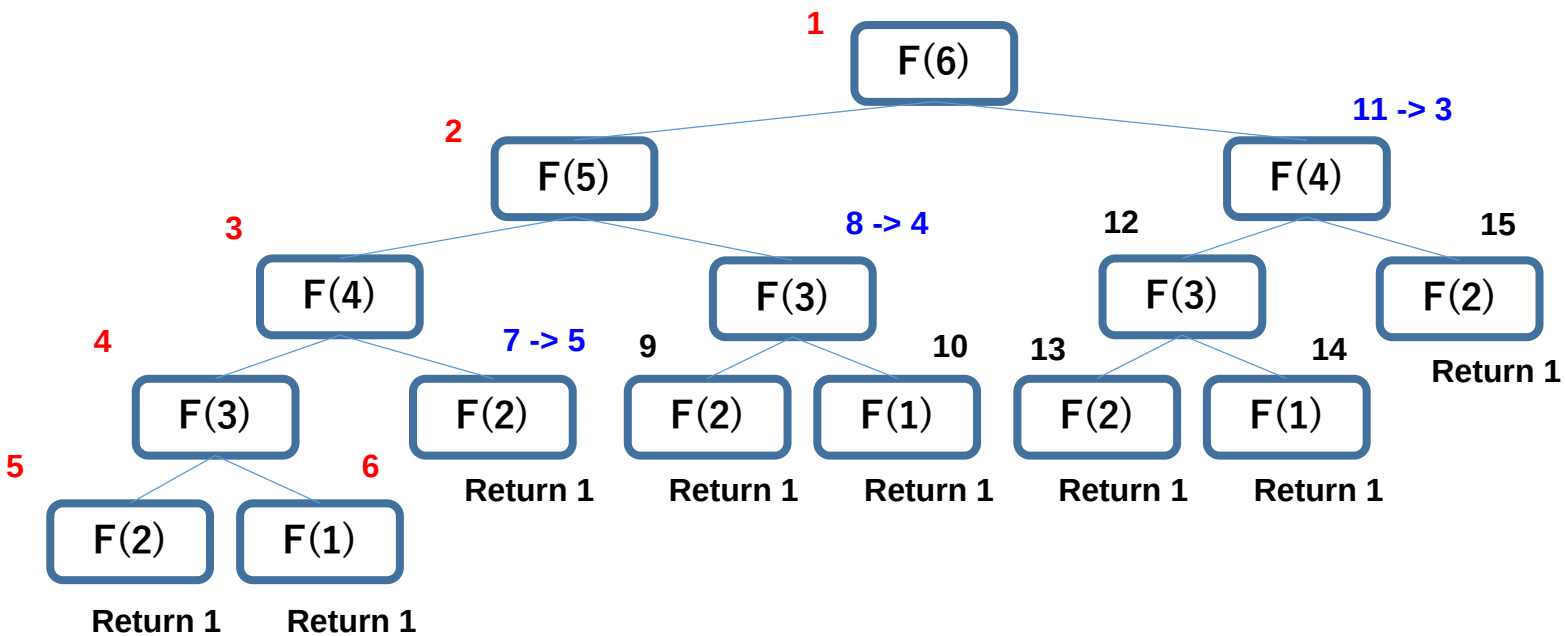


フィボナッチ数列(Fibonacci number)

- このフィボナッチ数は f(n) = f(n-1) + f(n-2) という式で計算することができます。このとき、f(1) = 1、f(2) = 1です。
- 関数自身が自分自身で定義されており、そのまま再帰関数の形になっていることに注目してください。
- n=6 の場合のフィボナッチ数の計算の図のようになります。ここでは、実際のプログラムではなくて数学の関数のような概念として計算の流れを説明しています。
- 1つの関数はれぞれ自分自身を2回呼び出していることに注意して下さい。またそれぞれの最後の呼び出しの部分では, n=1 または n=2 が成り立っていることが分かるかと思います。
- This Fibonacci number can be calculated with the formula f(n) = f(n−1) + f(n−2). Where, f(1) = 1 and f(2) = 1.
- Note that the function itself is defined by using itself. Then, it is directly in the form of a recursive function.
- The diagram of the Fibonacci number calculation for n=6 is shown previous page. Here, the calculation flow is described as a mathematical function-like concept.
- Note that each function calls itself twice. You can also see that the last call of each function end with n=1 or n=2.

メモ化再帰(memoized recursion)

- メモ化再帰 (memoized recursion)
  - 一回計算した値を保持しておいて再利用する仕組み
  - 計算の効率化を測れる
  - 計算時間と保持した値を持ってくる時間を比較する必要があります



フィボナッチ数列(Fibonacci number)

- この図では、関数の計算順を左肩に示しています。赤字で示したのが、ある引数での最初の呼び出しになります。
- ここで、n=3 や n=4 の時の答えを一度計算しておけば、次に必要になった場合にその答えを流用できることが分かります（青字で示した）。
- このように一度計算した値を保持し、次の計算の時に再利用することで計算の効率化を測れます。これを メモ化再帰と呼びます。

- In this figure, the order of computation is shown on the left shoulder. The first call with a specific argument is shown in red.
- Here, we already have calculated the answer for n=3 or n=4, then we can reuse that answer at the next time （shown in blue).
- In that way, once calculated values are stored, they can be reused for the next calculation, thus improving the efficiency of the calculation. This calls ″memoized recursion“
- Need to compare the time of calculation time with the time to bring the stored value

再帰呼び出しを用いたフィボナッチ数列

```c
/* header files */
#include<stdio.h>

/* functions */
int fibonacci(int n);

/* main */
int main(void) {

    int n;
    printf("Fibonacci number f(n) = f(n - 1) + f(n - 2).¥n");
    printf("Enter a positive integer n (1 <= n <= 45): ");
    scanf("%d", &n);

    printf("f(%d) = %d¥n", n, fibonacci(n));

    return 0;
}
```

```c
/*
the calculation of a Fibonacci number
 using recursion
*/
int fibonacci(int n) {

    if (n == 1 || n == 2) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Execution result

```
Fibonacci number f(n) = f(n - 1) + f(n - 2).
Enter a positive integer n (1 <= n <= 45): 45
f(45) = 1134903170
```

フィボナッチ数の計算を実際のプログラムにしたのが右上の関数です。
ここでは、n=1 と n=2 の時が終了条件(1を返す)となります。

Fibonacci series using recursive function call

```c
/* header files */
#include<stdio.h>

/* functions */
int fibonacci(int n);

/* main */
int main(void) {

    int n;
    printf("Fibonacci number f(n) = f(n - 1) + f(n - 2).¥n");
    printf("Enter a positive integer n (1 <= n <= 45): ");
    scanf("%d", &n);

    printf("f(%d) = %d¥n", n, fibonacci(n));

    return 0;
}
```

```c
/*
the calculation of a Fibonacci number
 using recursion
*/
int fibonacci(int n) {

    if (n == 1 || n == 2) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Execution result

```
Fibonacci number f(n) = f(n - 1) + f(n - 2).
Enter a positive integer n (1 <= n <= 45): 45
f(45) = 1134903170
```

The actual program for calculating Fibonacci numbers is in the upper right.
When n=1 and n=2, the termination conditions is satisfied (then, return 1).

配列の宣言、初期化と関数呼び出し

- メモ化再帰を行う時に利用できる
  - 既に計算した値を保持しておく

```c
#include <stdio.h>

#define ARRAY_SIZE 5

/* main */
int main(void) {
  // 配列の宣言(初期化なし)
  int a[ARRAY_SIZE];
  // 配列の初期化(ゼロ)
  int b[ARRAY_SIZE] = {};
  // 配列の初期化
  int c[ARRAY_SIZE] = {0, 1, 2, 3, 4};

  puts("1D array (no initialize)");
  func1(a, ARRAY_SIZE);
  puts("1D array (zero)");
  func1(b, ARRAY_SIZE);
  puts("1D array");
  func1(c, ARRAY_SIZE);
}
```

- 配列は基本的に初期化して使ってください。
- 配列はサイズを持たないので、関数にはサイズも渡す必要があります。
- 配列を関数に渡した場合、配列を書き換えると元の配列の値が書き換わります。
  - これを参照渡しと言います。(高度な内容)
- この振る舞いを利用して、計算した値を保持しておくことができます
- 初期値から書き換わっていないか判定するために、計算した値ではない値を初期値にすると良いでしょう。

```c
// 配列を受け取る関数
int func1(int ary[], int size)
{
  int i;
  for(i = 0; i < size; i++) {
    printf("%d: %d¥n", i, ary[i]);
  }
}
```

Declarations and initialization of an array and using as arguments

- Arrays can be used for ″memoized recursion“
  - Store values already calculated

```c
#include <stdio.h>

#define ARRAY_SIZE 5

/* main */
int main(void) {
  // 配列の宣言(初期化なし)
  int a[ARRAY_SIZE];
  // 配列の初期化(ゼロ)
  int b[ARRAY_SIZE] = {};
  // 配列の初期化
  int c[ARRAY_SIZE] = {0, 1, 2, 3, 4};

  puts("1D array (no initialize)");
  func1(a, ARRAY_SIZE);
  puts("1D array (zero)");
  func1(b, ARRAY_SIZE);
  puts("1D array");
  func1(c, ARRAY_SIZE);
}
```

- Arrays should be initialized before using
- Since arrays do not have a size, the size must also be passed to a function
- If an array is passed to a function, rewriting the element of the array will rewrite the element in the original array values
- This behavior can be used to keep the calculated values
- To determine if the initial value has been rewritten, you may use a value that is not the calculated value as the initial

```c
// 配列を受け取る関数
int func1(int ary[], int size)
{
  int i;
  for(i = 0; i < size; i++) {
    printf("%d: %d¥n", i, ary[i]);
  }
}
```

配列の宣言、初期化と関数呼び出し

- メモ化再帰を行う時に利用できる
  - 既に計算した値を保持しておく

- #defineで宣言された文字はプログラムがコンパイルされる前に置き換わります。
- プログラム全体の定数等の記述に使います。
- 配列は基本的に初期化して使ってください。
- 2次元配列を関数に渡すときは、1次のサイズを指定する必要があります。

```c
#include <stdio.h>

#define ARRAY_SIZE 5

/* main */
int main(void) {

  // 2次元配列の初期化
  int aa[][ARRAY_SIZE] = {{0, 1, 2, 3, 4},
                          {5, 6, 7, 8, 9}};

  puts("2D array");
  func2(aa, 2);
}
```

```c
// 配列を受け取る関数(2次元)
int func2(int ary[][ARRAY_SIZE], int size)
{
  int i,j;
  for(i = 0; i < size; i++) {
    for(j = 0; j < 5; j++) {
      printf("%d-%d: %d¥n", i, j, ary[i][j]);
    }
  }
}
```

Declarations and initialization of an array and using as arguments

- Arrays can be used for ″memoized recursion“
  - Store values already calculated

- The characters declared in #define are replaced before the program is compiled
- Used to describe constants, etc. for the entire program
- When passing a two-dimensional array to a function, the size of the first order must be specified

```c
#include <stdio.h>

#define ARRAY_SIZE 5

/* main */
int main(void) {

  // 2次元配列の初期化
  int aa[][ARRAY_SIZE] = {{0, 1, 2, 3, 4},
                          {5, 6, 7, 8, 9}};

  puts("2D array");
  func2(aa, 2);
}
```

```c
// 配列を受け取る関数(2次元)
int func2(int ary[][ARRAY_SIZE], int size)
{
  int i,j;
  for(i = 0; i < size; i++) {
    for(j = 0; j < 5; j++) {
      printf("%d-%d: %d¥n", i, j, ary[i][j]);
    }
  }
}
```