明治大学理工学部情報科学科

ソフトウェア実習 レポート

5章 再帰

提出日 2025 年 10 月 25 日

3 年 15組 91番 157R257501

SEO BEOMGYO

はじめに

本課題の目的は、再帰の基本形と停止条件の設計を通じて計算手続を構造化し、入出力仕様に適合する小規模プログラム群を整備することである。

課題1

```c
#include <stdio.h>

int fact(int n)
{
    if (n == 0) return 1;

    return n * fact(n - 1);
}
```

- 目的：定義 $n! = n \times (n-1)!$ と $0! = 1$ に従い、再帰関数 fact を用いて階乗を計算する。
- 実装要点：基本事例は $n = 0$ で 1 を返し、その他は $n \times \mathrm{fact}(n-1)$ を返す単純再帰とした。
- テストと実行結果
  - 入力：内部で $n = 9$ を設定。
  - 出力：“fact(9) = 362880” を標準出力へ表示する。
  - 検証：$9! = 362880$ と一致し、32bit int では $13!$ 以上で桁あふれが起こり得る。

課題2

```c
#include <stdio.h>
void hanoi(int n, char *from, char *to, char *tmp);

int main(void)
{
    hanoi(3, "a", "b", "c");
    return 0;
}

void hanoi(int n, char *from, char *to, char *tmp)
{
    if (n == 0) return;

    hanoi(n - 1, from, tmp, to);      // 一旦、上の n-1 枚を補助柱 tmp へ移す（from→tmp）
    printf("move disk %d from %s to %s\n", n, from, to);  // 最大の 1 枚を目的柱へ移動
    hanoi(n - 1, tmp, to, from);      // 補助柱の n-1 枚を目的柱へ移す（tmp→to）
}
```

- 目的：hanoi(n, from, to, tmp) により、n 段の塔を from から to へ移す手続を記述する。
- 実装要点：停止条件は $n = 0$ で return、再帰分解は hanoi(n-1, from, tmp, to)

→ 最大円盤の移動 → hanoi(n-1, tmp, to, from) の三段から成る。

- テストと実行結果
  - 入力：hanoi(3, ”a”,”b”,”c”) を呼出。
  - 出力：移動ログ 7 行が既定順で出力される。
  - 検証：総手数は $2^n - 1$ であり $n = 3$ なら 7 と一致する。

課題3

```
3   void print_base(unsigned int x, int b)
4   {
5       if (x < (unsigned)b) {
6           putchar('0' + (int)x);
7           return;
8       }
9       print_base(x / (unsigned)b, b);
10      putchar('0' + (int)(x % (unsigned)b));
11  }
```

- 目的：x と b を受け取り、x を b 進で表示する再帰的印字手続を作成する。
- 実装要点：条件 $x < b$ なら 1 桁を印字し、そうでなければ print_base(x/b, b) を先行させてから $xmodb$ を出力することで上位桁から並べる。
- テストと実行結果
  - 入力：x=17, b=3 を順に入力。
  - 出力：122 を出力する。
  - 検証：$17 = 1 \times 3^2 + 2 \times 3 + 2$ より表示値は妥当である。

課題4

```
3   long long fib(int n)
4   {
5       if (n == 0) return 0;
6       if (n == 1) return 1;
7
8       // 再帰: f(n) = f(n-1) + f(n-2)
9       return fib(n - 1) + fib(n - 2);
10  }
```

- 目的：定義 $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$ をそのまま再帰で実装し f_n を求める。
- 実装要点：基本事例 n∈{0,1} を分岐し、他は fib(n-1)+fib(n-2) を返す単純再帰とした。

- テストと実行結果
    - 入力：例として n=10 を入力。
    - 出力：55 を出力する。
    - 検証：中間結果の重複計算が指数的増大を招き、n が大きいと実行時間が増大する。

課題5

```
5   long long fib_fast(int n)
6   {
7       // 反復で高速に計算（重複計算なし）
8       if (n == 0) return 0;
9       if (n == 1) return 1;
10      long long a = 0, b = 1;
11      for (int i = 2; i <= n; i++) {
12          long long c = a + b;
13          a = b;
14          b = c;
15      }
16      return b;
17  }
```

- 目的：課題4が遅い理由を示し、反復で中間状態を更新する高速版 fib_fast を示す。
- 実装要点：変数 a,b を用い $i=2$ から n まで $c=a+b, a=b, b=c$ と更新して重複計算を排除することで時間計算量を線形化した。
- テストと実行結果
    - 入力：n を標準入力で受け取る。
    - 出力：f_n を即時に出力する。
    - 検証：単純再帰と比較して関数呼出回数が削減され、スタック使用も抑制される。

課題6

```c
void generate(int level)
{
    if (level >= n) {
        for (int i = 0; i < n; i++) {
            printf("%d ", a[i]);
        }
        putchar('\n');
        return;
    }

    for (int v = 1; v <= n; v++) {
        a[level] = v;          // level 番目の値を記録
        generate(level + 1);   // 次の桁を生成
    }
}
```

- 目的：1..n から長さ n の列を重複可で全列挙し、計 n^n 行を出力する。
- 実装要点：a[level] に値 v=1..n を割当てつつ generate(level+1) に降下し、level==n で 1 行出力する。
- テストと実行結果
  - 入力：n=2。
  - 出力：“1 1” “1 2” “2 1” “2 2” を出力する。
  - 検証：出力行数は n^n で n=2 なら 4 行となる。

課題7

```c
void generate(int level)
{
    if (level >= n) {
        for (int i = 0; i < n; i++) {
            if (i) putchar(' ');
            printf("%d", perm[i]);
        }
        putchar('\n');
        return;
    }

    for (int v = 1; v <= n; v++) {
        if (used[v]) continue;   // すでに使った数はスキップ
        used[v] = true;          // 使用開始
        perm[level] = v;         // level 番目に配置
        generate(level + 1);     // 次の位置へ
        used[v] = false;         // 戻って未使用に戻す
    }
}
```

- 目的：1..n の順列 $n!$ 通りを全て出力する。
- 実装要点：used[v] で使用状況を管理し、perm[level]=v を置いて次段へ進み、復帰時に used[v]=false で巻き戻す典型的バックトラックを用いる。
- テストと実行結果
  - 入力：n=3。
  - 出力：6 通りの順列を 1 行 1 通りで出力する。
  - 検証：行数は 3! = 6 と一致する。

課題8

```
3   long long partition(int n, int x)
4   {
5       if (n == 0) return 1;
6
7       if (n < 0 || x == 0) return 0;
8
9       return partition(n, x - 1) + partition(n - x, x);
10  }
11
12  long long p(int n)
13  {
14      return partition(n, n);
15  }
```

- 目的：正の整数 n を和で表す方法の数 $p(n)$ を partition(n, x) により計算する。
- 実装要点：“x を使わない”場合の partition(n, x-1) と “x を使う”場合の partition(n-x, x) を合算し、n==0 で 1、n<0 または x==0 で 0 を返す。
- テストと実行結果
  - 入力：n=5。
  - 出力：“p(5) = 7” を表示する。
  - 検証：例示される 7 通りの分割と一致し、p(n)=partition(n,n) で求まる。

課題9

```
// 位置の差 == 値の差になっていないか衝突チェック
static bool ok_place(int level, int v)
{
    // level に v を置いても良いかを確認
    for (int k = 0; k < level; k++) {
        int dl = level - k;          // 位置の差
        int dv = v - perm[k];        // 値の差
        if (dl < 0) dl = -dl;
        if (dv < 0) dv = -dv;
        if (dl == dv) return false;
    }
    return true;
}

void generate(int level)
{
    if (level >= n) {
        for (int i = 0; i < n; i++) {
            if (i) putchar(' ');
            printf("%d", perm[i]);
        }
        putchar('\n');
        count++;
        return;
    }

    for (int v = 1; v <= n; v++) {
        if (used[v]) continue;
        if (!ok_place(level, v)) continue;
        used[v] = true;
        perm[level] = v;
        generate(level + 1);
        used[v] = false;
    }
}
```

- 目的：任意の二位置 i,j に対し $|i-j| \neq |a_i - a_j|$ を満たす順列のみを出力し、解の個数を集計する。
- 実装要点：ok_place(level, v) で既設置要素 perm[k] に対する $|\text{level} - k|$ と $|v - \text{perm}[k]|$ を比較し、等しければ衝突として枝刈りする。
- テストと実行結果
    - 入力：n を 1-9 の範囲で入力する。
    - 出力：条件を満たす順列を列挙し、最後に “nの解の個数 ： c” を表示する。
    - 検証：生成途中で違反を検出して打切るため、探索木の大部分を削減できる。

考察

- フィボナッチの単純再帰は部分問題を重複評価するため指数的に遅く、反復更新やメモ化への置換が実用上必須である。