1　再帰処理

1.1　再帰処理の考え方

クイックソートのように、自分自身を呼び出すような処理のことを言います。C 言語で言えば、ある関数が自分自身を呼び出して処理を行なう場合、その関数呼び出しは再帰的であると言います。

再帰処理を正しく行なうためには、ただ単に自分自身を呼び出すだけではいけません。呼び出される関数はある種の条件判定を行い、必要に応じて再帰呼び出しを行なうかどうかを判断します。こうしないと、無限に再帰呼び出しが行なわれてしまい、プログラムが止まらなくなってしまいます。

再帰呼び出しの説明には、よく階乗の計算が例題として示されます。これは、階乗の計算が次のように再帰的に記述できるためでしょう。

1.2　階乗の計算アルゴリズム (再帰処理)

関数 factorial(int n)

1　もし n が 0 なら 1 を返す

2　そうでなければ、n*factorial(n-1) を返す

上記では、n の階乗を求めるためには、n-1 の階乗の値に n を掛ければよいことを利用して、再帰的に求めています。ただし n が 0 のときには、何も計算せずに、定義に従って 0 の階乗の値である 1 を返します。

1.3　factorial.c

```
/*
 factorial.c
 階乗を再帰で求める。
*/

#define BUFSIZE 256
#include <stdio.h>

int getint(void);
```

```
int factorial(int n);

int main(void)
{
  int n;
  while((n = getint()) >= 0){
    printf("%d! = %d\n\n", n, factorial(n));
  }
  return 0;
}

int factorial(int n)
{
 if(n == 0){
   return 1;
 }else{
   return n * factorial(n-1);
 }
}

int getint(void)
{
  char linebuf[BUFSIZE];
  int  n;

  if(fgets(linebuf, BUFSIZE, stdin) != NULL){
    if(sscanf(linebuf, "%d", &n) <= 0){
      n = -1;
    }
  }else{
    n = -1;
  }
  return n;
}
```

1.4 factorial の使い方

```
$ ./factorial
```

```
7
7!=5040
10
10!=3628800
-1（負数の入力で終了）

$
```

1.5 再帰処理を使う意味

ソフトウェア開発の立場から言うと、再帰を使う意味は、再帰的な定義をそのままプログラムとして書き出すことで、プログラムの可読性を高め、プログラムの生産性を向上させることにあります。したがって再帰が効率的に利用されるのは、アルゴリズムが再帰的に定義されており、繰り返し処理で記述するよりも再帰的な記述の方が自然な記述となる場合です。
C によるソフトウェア開発の基礎　103 頁　小高　知宏著　オーム社

2 6174 の不思議　Kapreker Number

$1111, 2222, \cdots, 9999$ 以外の任意の 4 桁の数 x を考えてください。x の数字の並べ方を変えて最大値と最小値を作り、その差を $g(x)$ とします。このとき

$$x \rightarrow\ g(x) \rightarrow\ g(g(x)) \rightarrow\ g(g(g(x))) \rightarrow \cdots$$

と続けていくとどのようになるかを確かめるプログラムを作成してください。

ただし、3 桁以下の数に対しては上位桁に 0 を補って y を求めます。例えば

$$g(999) = g(0999) = 9990 - 0999 = 8901$$

となります。
「C による探索プログラム　伊庭斉志著　オーム社)　16 頁

2.1 m6174.c

```
/*
```

```
  C による探索プログラム
  16 頁
  練習問題 2.1
  Kaprekar Number カプレカ数
  m6174.c
*/

#include <stdio.h>

int func(int x);

int main()
{
  int i;
  int x;

  printf("4 桁の数字を入力してください。->");
  scanf("%d", &x);

  func(x);

  return 0;
}

int func(int x)
{
  int i;
  int j;

  int maximum;
  int minimum;

  int keta[4];

  int max[4];
  int min[4];

  int work;

  if( x != 6174){
```

```
keta[3] = x / 1000;
keta[2] = (x - keta[3] * 1000) / 100;
keta[1] = (x - keta[3] * 1000 - keta[2] * 100 ) /10;
keta[0] = x - keta[3] * 1000 - keta[2] * 100 - keta[1] * 10;

printf("入力された数字は、");
for ( i = 3; i >= 0; i--){
  printf("%d", keta[i]);
}
printf("\n");

for ( i = 3; i >= 0; i--){
  max[i] = keta[i];
  min[i] = keta[i];
}

for( j = 3; j > 0; j--){
  for ( i = 3; i > 0; i--){
    if(max[i] < max[i -1]){
      work = max[i];
      max[i] = max[i - 1];
      max[i - 1] = work;
    }
  }
}

printf("最大数は、");
for ( i = 3; i >= 0; i--){
  printf("%d", max[i]);
}
printf("\n");

for( j = 3; j > 0; j--){
  for ( i = 3; i > 0; i--){
    if(min[i] > min[i - 1]){
      work = min[i];
      min[i] = min[i - 1];
      min[i - 1] = work;
    }
  }
```

```
    }

    printf("最小数は、");
    for ( i = 3; i >= 0; i--){
      printf("%d", min[i]);
    }
    printf("\n");

    maximum = max[3] * 1000 + max[2] * 100 + max[1] * 10 + max[0];
    minimum = min[3] * 1000 + min[2] * 100 + min[1] * 10 + min[0];

    printf("%d - %d = %d\n", maximum, minimum, maximum - minimum);

    func(maximum - minimum);
  }
}
```

3 ハノイの塔 (再帰の話)

ハノイの塔ゲームとは、次のようなものです。

「3 本の棒、a,b,c があり、棒 a に、中央に穴があいた n 枚の円盤が、直径の大きいものを下にして順序よく積まれています。これを棒 b に積み変えるのがゲームです。ただし、円盤は 1 度に 1 枚の円盤しか移動してはいけません。また、移動中に、小さな円盤の上に、大きな円盤がるようなことがあってはいけません。

```
/*
  ハノイの塔問題の再帰解
  hanoi.c
*/

#include <stdio.h>

void hanoi(int n, char a, char b, char c)
{
  if(n > 0){
    hanoi(n - 1, a, c, b);
    printf("%d 番の板を %c から %c に移動\n", n, a, b);
    hanoi(n - 1, c, b, a);
```

```
  }
}

int main()
{
  int n;
  printf("円板の枚数 ? ");
  scanf("%d", &n);
  hanoi(n, 'a', 'b', 'c');

  return 0;
}
```

「C 言語」(河西朝雄著　ナツメ社)　120 頁

4　AI による大規模データ処理入門

4.1　発見的な探索

横型探索と縦型探索は、あらかじめ決められた順番に従って網羅的に探索を行なう手法でした。これらの探索手法は、問題に依存しない汎用的手法ですが、探索の効率がよくありません。たとえば迷路の探索の例で言えば、ゴールに近づくかどうかに関係なく、常に決められた順番で探索が進められます。また、横型探索と縦型探索では、解が見つかって探索が終了したとしても、得られた探索結果が良い解であると言う保証は有りません。たとえば迷路の例で言えば、ゴールまでたどり着いたとしても、その道筋が最短経路になっているという保証はありません。
問題の性質を利用することで、より効率的に探索を行う方法を示します。
AI による大規模データ処理入門　71 頁　小高　知宏著　オーム社

4.2　最良優先探索プログラム

```
/*
  最良優先探索プログラム
  best.c
*/
```

```
#include <stdio.h>

#define LIMITL 256
#define TRUE 1
#define START 1
#define GOAL 999

struct node{
  int    nodeid;
  int    parentid;
  double value;
};

struct node openlist[LIMITL];
int         openlistep = 0;
struct node closedlist[LIMITL];
int         closedlistep = 0;
double      h [] = {0, 5.7, 4.5, 2.8, 4.1, 2, 4, 4, 0};

void initlist();
void printlist();
void expand(int id);
void movetofirst();
void removefirst();
int  check(int id);
void printroute(int id);
void sortopenlist();
int  cmp(const void *a, const void *b);

int main()
{
  initlist();
  printlist();

  while(TRUE){
    if(openlistep == 0){
      printf("ゴールは見つかりませんでした\n");
      break;
    }
```

```
    if(openlist[0].nodeid == GOAL){
      printf("\n ゴールを見つけました\n");
      printf("%d[%d]", openlist[0].nodeid,
                       openlist[0].parentid);
      printroute(openlist[0].parentid);
      break;
    }

    expand(openlist[0].nodeid);

    movetofirst();

    sortopenlist();

    printlist();
  }

  return 0;
}

void sortopenlist()
{
  qsort(openlist, openlistep, sizeof(struct node), cmp);
}

int cmp(const void *a, const void *b)
{
  struct node *x, *y;
  x = (struct node *)a;
  y = (struct node *)b;
  if((x->value) < (y->value)){
    return -1;
  }else{
    if((x->value) > (y->value)){
      return 1;
    }else{
     return 0;
    }
  }
```

```
}

void printroute(int id)
{
  int i;

  for(i = 0; i < closedlistep; ++i){
    if(closedlist[i].nodeid == id){
      printf("<-%d[%d]", closedlist[i].nodeid,
                         closedlist[i].parentid);
      break;
    }
  }

  if(closedlist[i].parentid != 0){
    printroute(closedlist[i].parentid);
  }

  printf("\n");
}

int check(int id)
{
  int i;
  int res = 0;

  for(i = 0; i < openlistep; ++i){
    if(openlist[i].nodeid == id){
      res = TRUE;
    }
  }

  for(i = 0; i < closedlistep; ++i){
    if(closedlist[i].nodeid == id){
      res = TRUE;
    }
  }

  return res;
}
```

```
void removefirst()
{
  int i;
  for(i = 0; i < openlistep; ++i){
    openlist[i] = openlist[i + 1];
  }
  --openlistep;
}

void movetofirst()
{
  closedlist[closedlistep++] = openlist[0];
  removefirst();
}

void expand(int id)
{
  int tree[][5] = {
    {1, 2, 4},
    {2, 6},
    {3, 5, 999},
    {4, 3},
    {6, 5, 7},
    {5, 7, 999},
    {0}
  };

  int i = 0;
  int j;

  while(tree[i][0] != 0){
    if(tree[i][0] == id){
      for(j = 1; tree[i][j] != 0; ++j){

        if(check(tree[i][j]) != TRUE){
          openlist[openlistep].nodeid = tree[i][j];
          openlist[openlistep].parentid = id;
          openlist[openlistep++].value = h[tree[i][j]];
        }
```

```
      }
      break;
    }
    ++i;
  }
}

void initlist()
{
  openlist[0].nodeid = START;
  openlist[0].parentid = 0;
  ++openlistep;
}

void printlist()
{
  int i;

  printf("\n オープンリスト    ");
  for(i = 0; i < openlistep; ++i){
    printf("%d[%d, %.1lf],", openlist[i].nodeid,
                             openlist[i].parentid,
                             openlist[i].value);
  }
  printf("\n");

  printf("クロースドリスト  ");
  for(i = 0; i < closedlistep; ++i){
    printf("%d[%d],", closedlist[i].nodeid,
                      closedlist[i].parentid);
  }
  printf("\n");
}
```

4.3 使い方

```
./best
```