グラフとアルゴリズムとプログラムのやさしいおはなし

渡邉敏正

2022 年 3 月 24 日

第 11 回

1　関数を使ってみよう

第 10 回でプログラム compo.c を示しました。これは第 6 回で示したアルゴリズム **compo(G, s)** を第 7 回〜第 9 回で行った準備に基づいて作成したものです。プログラム作成では、実行結果を表示する操作を追加し、とにかく正しく動くプログラムを作ることに重点を置きました。

しかしながら、プログラムとしてはより洗練した形で読みやすく、また改良や拡張などの今後の扱いが容易な形であることが望ましい姿です。それで今回は、あまり専門的な事項に深入りせずこれまで説明してきた事項を使ってできる範囲内で、プログラム compo.c を少しでも読みやすくできるだけ簡潔な記述に改良することに挑戦してみようと思います。

具体的には、新たに関数という仕組みに着目します。プログラムに関数を取り込むことで全体の筋道がよく見えて理解しやすい記述になることを示そう、というのが今回の目標です。

1.1　アルゴリズム compo(G, s) とプログラム compo.c のおさらい

1.1.1　アルゴリズム compo(G, s)

グラフ G における基点 s からの距離という数値を導入し、距離 0 の点の集合を $A_0 = \{s\}$ として、距離 1 の点の集合 A_1、距離 2 の点の集合 $A_2, \cdots$ 、と距離集合 $A_i\,(i = 1, \cdots)$ を求めていくことで、グラフ G 上の基点 s から連結な点をすべて求めるアルゴリズム **compo(G, s)** を第 6 回で作成しました。アルゴリズムは $N_0 = A_0$ として、$N_1 = A_1, \cdots, N_k = A_k$ なる集合 $N_1, \cdots, N_k$ を構成していきます。以下が提案したアルゴリズムです。

compo(G, s)

/* G が 1 点 s のみの場合は 1 回目の 3.3 で $N_1 = \emptyset$ となり、2 回目の 3.1 において $N_1 = \emptyset$ で Step4 に進み終了します。したがって、以下では G が 2 点以上を含むとして説明します */

Step1.　$k \leftarrow 1$ とする;

Step2.　$N_0 \leftarrow \{s\}$, $R \leftarrow N_0$ とする;　/* $A_0 = \{s\}$ であり、R は最終的には $A(s)$ を格納します */

Step3.

3.1.　$N_{k-1} \neq \emptyset$ である間は、以下の 3.2〜3.4 を実行する; /* $N_{k-1} = \emptyset$ ならば Step4 進む */

3.2.　$N_k \leftarrow \emptyset$ とする; /* 各 k について、N_k は A_k に入るべき要素を保持します */

3.3.　N_{k-1} のすべての点 $u_1, \cdots, u_r$ に対して、$i = 1, \cdots, r$ の順に、各 u_i について以下の拡大操作を行う;

(拡大操作)

- u_i に隣接する点が存在しないときは何もしない;
- u_i に隣接する点が $v_1, \cdots, v_d\,(d \geq 1)$ のとき、$j = 1, \cdots, d$ の順に、各 v_j について以下の (i) または (ii) を実行する;

  (i)　$k = 1$ のとき、v_j を N_k に加える; /* 図 113 参照 */

  (ii)　$k \geq 2$ のとき、$v_j \notin R \cup N_k$ ならば v_j を N_k に加え、$v_j \in R \cup N_k$ ならば何もしない;[*1]　/* 図 114 参照 */

3.4.　もし $N_k \neq \emptyset$ ならば $R \leftarrow R \cup N_k$ とする。k の値を 1 だけ増やして 3.1 に戻る;

Step4.　終了する; /* $N_{k-1} = \emptyset$ */

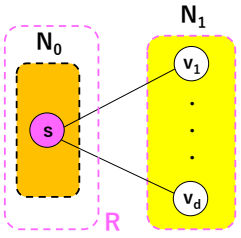

図 113　$N_0 = \{s\}$、および $s\,(= u_1)$ に隣接する点 v_j の集合 N_1；まず $R \leftarrow N_0$ と初期設定し、N_1 が空でない集合として確定したならば $R \leftarrow R \cup N_1$ と更新します

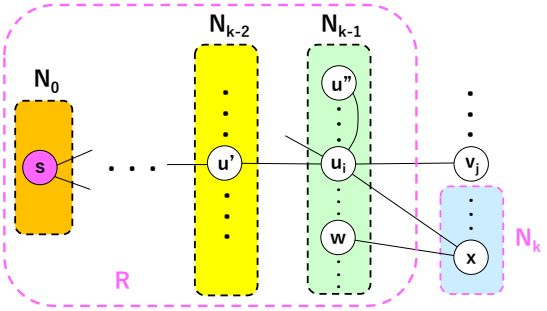

図 114　N_{k-1} の点 u_i に隣接する点 $u' \in N_{k-2} \subset R$, $u" \in N_{k-1} \subset R$, $x \in N_k$ および $v_j \notin R \cup N_k$ (実質的には、$v_j \notin N_{k-2} \cup N_{k-1} \cup N_k$)；このとき、$N_k \leftarrow N_k \cup \{v_j\}$ と更新し、最終的に N_k が空でない集合として確定したならば $R \leftarrow R \cup N_k$ なる更新もします

N_k, R に着目した **compo(G, s)** の大まかな流れ図が図 115 です。アルゴリズムの流れが見えると思います。

図 116 のグラフ G_4 と基点 $s = 4$ に対して **compo(G, s)** を実行した場合に、A_0, A_1, A_2, A_3 がそれぞれ N_0, N_1, N_2, N_3 として順次求められていく様子を図 117〜図 120 に示しました。アルゴリズムの動作を理解する助けにしてください。

[*1] v_j が $R \cup N_k$ に含まれるか否かは、詳細に言えば、v_j が $N_{k-2} \cup N_{k-1} \cup N_k$ 含まれるか否かの判定です。

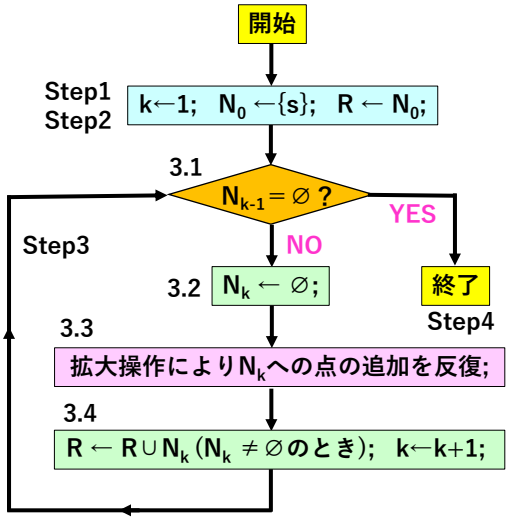

図 115　N_k, R に着目した $\mathbf{compo(G, s)}$ の流れ図

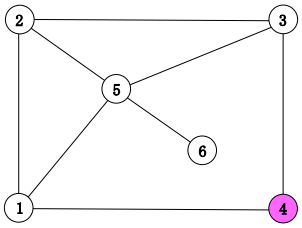

図 116　グラフ $G_4 = (V_4, E_4)$ と基点 $s = 4$

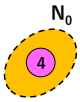

図 117　Step2 終了時の状況 (R は省略)

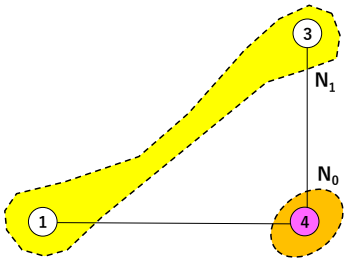

図 118　1 回目の 3.3 終了時の状況 (R は省略)

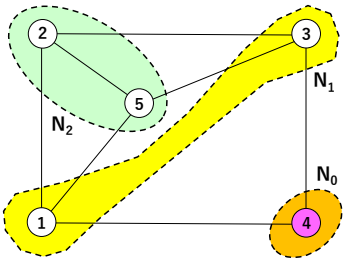

図 119　2 回目の 3.3 終了時の状況 (R は省略)

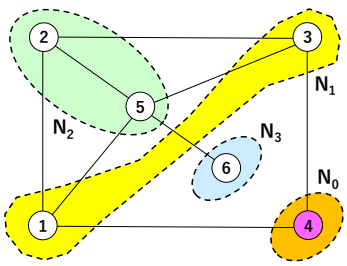

図 120　3 回目の 3.3 終了時の状況 (R は省略)

1.1.2 プログラム compo.c

今回の改良対象である compo.c のプログラム自体は長いので、それを掲載するかわりに枠組みを示しておきます。compo.c は以下のような枠組みになっています。このプログラムを例として、新たに紹介する関数について説明していきます。

(**compo(G, s)** のプログラム全体の枠組み)

```
// Computing vertices connected to a specified one s in a graph
#include <stdio.h>
#define n 6
#define r 4
#define s 4

int main(void)
{
    int i, j, k, sum;
    int A[n+1][n+1]={
        {0,0,0,0,0,0,0},{0,0,1,0,1,1,0},{0,1,0,1,0,1,0},{0,0,1,0,1,1,0},
        {0,1,0,1,0,0,0},{0,1,1,1,0,0,1},{0,0,0,0,0,1,0}
    };
    int N[r+1][n+1];
    int R[n+1];

    //タイトル表示

    // 初期化
    for (i=0; i<=r; i++){
        for (j=0; j<=n; j++){
            N[i][j]=0;
        }
    }
    for (i=0; i<=n; i++){
        R[i]=0;
    }

    // *** Step1 ***
    k=1;

    // *** Step2 ***
    N[0][s]=1; N[0][0]=N[0][0]+1;
    R[s]=1; R[0]=R[0]+1;

    // *** Step3  拡大操作の反復 ***
        // 3.1 反復の判定（反復制御）
        // 3.2 N_k の初期化
        // 3.3 拡大操作
        // 3.4 N_k を R に追加；k を 1 だけ増加

    // ***** 結果の表示 *****
    // 点の次数の計算
    // 隣接行列の表示
```

```
    // 距離集合 N_0, ..., N_{r-1}, N_r の表示；
    // 点 s と連結な点の集合 R の表示

    // **** Step4 ***
    return 0;
}
```

1.2 「隣接行列の表示」と「距離集合の表示」の操作の類似性

「結果の表示」部分は **compo(G, s)** のプログラム作成で追加した箇所です。その中の

隣接行列の表示、　距離集合 N_0, ..., N_{r-1}, N_r の表示

なる 2 つの表示操作に着目してみましょう。この部分のプログラム記述は以下のようになっています。第 10 回で示した compo.c の記述から抜き出してきました。行番号はそのままにしています。

```
75      // 隣接行列の表示
76      printf("\n***** 隣接行列（左端は各点の次数） *****\n");
77      for (i=1; i<=n; i++){
78          for (j=0; j<=n; j++){
79              if (j==0){
80                  printf("A[%d][%d]=%d: ",i, j, A[i][j]);
81              }
82              else if (A[i][j]==1){
83                      printf("%d ",j);
84              }
85          }
86          printf("\n");
87      }


88      // s からの距離が k>=0 の点の集合 N_0, ..., N_{r-1}, N_{r}の表示；
89      // N_{r}は空集合です
90      printf("\n***** 点 %d からの距離が k>=0 の点の集合 N[0],...,N[%d]
                                        （左端は要素数） *****\n", s, r);
91      for (i=0; i<=r; i++){
92          for (j=0; j<=n; j++){
93              if (j==0){
94                  printf("N[%d][%d]=%d: ", i, j, N[i][j]);
95              }
96              else if (N[i][j]==1){
97                  printf("%d ", j);
98              }
99          }
100         printf("\n");
101     }
```

- 隣接行列の表示は 2 次元配列 A における要素 A[i][j] の $1 \leq i \leq n;\ 0 \leq j \leq n$ についてであり、距離集合 N_k の表示は 2 次元配列 N における要素 N[i][j] の $0 \leq i \leq r;\ 0 \leq j \leq n$ についてです。
- 77 行目〜87 行目と 91 行目〜101 行目を比較してみると一見してよく似ている操作であることがわか

ると思います。

- 実際、これらの操作の違いは i の反復が $1 \leq i \leq n$ と $0 \leq i \leq r$ である、表示の記号が `A[i][0]` と `N[i][0]` である、および `if` 文の判定条件が `A[i][j]==1` と `N[i][j]==1` であるという 3 点だけです。

このような類似した操作を一つの操作としてまとめて、これをそれぞれで利用することを考える、というのが関数の出発点です。

1.3 関数の定義と実行

そこで以下のような操作の記述を考えてみます。名前を `display_2d` としておきます。正確に言えば、以下のような記述が**関数の定義** (function definition) で、`display_2d` が**関数の名前** (function name) です。通常、

「以下が関数 `display_2d` の定義です」

という表現をします。なお、記述の中に出てくる配列添字 `n` や `r` はとりあえずはある定数と考えておいてください。

```
void display_2d(const int M[][n+1], int p1, int p2, int q1, int q2, char b)
{
    int i, j;

    for (i=p1; i<=p2; i++){
        for (j=q1; j<=q2; j++){
            if (j==0){
                printf("%c[%d][%d]=%d: ", b, i, j, M[i][j]);
            }
            else if (M[i][j]==1){
                printf("%d ", j);
            }

        }
        printf("\n");
    }
}
```

- 関数 `display_2d` は、整数型 2 次元配列 `M[][n+1]` の第 $p1$ 行から第 $p2$ 行までの各行 (i 行とする) について、第 $q1$ 列から第 $q2$ 列までの間にある `M[i][j]==1` なる点 j を横一列に表示します。ただし、各行の最初だけは `M[i][0]` の値を `b[i][0]=(`**値**`)` の形で表示します。以下に $q1 = 0$ の場合の具体的な表示形式を示します。

`b[p1][0]=`(値) ... (`M[p1][j]==1` なる点 `j` $(q1 \leq j \leq q2)$) ...
⋮ ⋮
`b[i][0]=`(値) ... (`M[i][j]==1` なる点 `j` $(q1 \leq j \leq q2)$) ...
⋮ ⋮
`b[p2][0]=`(値) ... (`M[p2][j]==1` なる点 `j` $(q1 \leq j \leq q2)$) ...

- char b は A[i][j] や N[i][j] などの配列名を指定します。例えば b ← 'A' と設定すると A[i][j]= の形の表示となり、b ← 'N' と設定すると N[i][j]= の形の表示となる、という具合です。文字を指定する場合は 'A' のように文字の前後を ' で囲みます。
- なお、const int M[][n+1] の const は

  M[][n+1] の要素は変更禁止である

  ということを示します。display_2d の実行中に M[][n+1] の要素が変更されないようにするためです。
- すぐにわかると思いますが、

  配列 M ← 配列 A;　p1 ← 1;　p2 ← n;　q1 ← 0;　q2 ← n;　表示文字 b ← 'A'

  と設定して display_2d を実行すれば「隣接行列の表示」の操作になります。一方、

  配列 M ← 配列 N;　p1 ← 0;　p2 ← r;　q1 ← 0;　q2 ← n;　表示文字 b ← 'N'

  と設定して display_2d を実行すれば「距離集合 N_k の表示」の操作になります。
- このように display_2d は 2 次元配列を表示させる操作の一つの雛形で、関数名の右のカッコ内に記載している配列や変数 (これらは、一般に**パラメータ**あるいは**仮引数**[*2](parameter) とよばれます) に具体的な配列や値などを設定すれば設定した配列の要素を表示する機能を持っており、様々な状況で使うことができます。パラメータ (仮引数) に設定された具体的な配列や値のことを**実引数**[*3](argument) とよびます。毎回似たような操作を記述しなくても、display_2d に具体的な配列や値などを設定すればいいことになりますので、その便利さは皆さんも理解できると思います。

注意 11.1 関数のカッコ内に記載の仮引数[*4]には必ずデータ型を明示しなければなりません。例えば、int M[][n+1] は整数型 2 次元配列を表し、int p1 は整数型変数を表しています。なお、2 次元配列に関しては第 1 番目の [ ] 内に要素数の記載は不要です。2 番目以降は [n+1] のように要素数を記載します。1 次元配列の場合も、例えば M[ ] のように [ ] 内に要素数の記載は不要です。

では、display_2d の仮引数に具体的な配列や値などを設定するにはどうすればいいのでしょうか。以下でこのことについて説明します。

1.4 関数の呼び出しとデータの受渡し

display_2d の仮引数に具体的な配列や値などを設定して実行させる方法について説明します。

1.4.1 概要

まず、プログラムの中で main の前 (ただし, #include..., #define... の後) に関数 display_2d 全体を記述しておきます。そして、「隣接行列の表示」については

```
display_2d(A, 1, n, 0, n, 'A')
```

を、「距離集合 N_k の表示」については

[*2] 読み方は「かりひきすう」です。
[*3] 読み方は「じつひきすう」です。
[*4] 以後、実引数との対応で「仮引数」を使います。

```
display_2d(N, 1, n, 0, n, 'N')
```

をプログラムの中 (例えば main の中) に記述します。これで仮引数に上述の配列を設定したことにあるいは値を代入したことになり、設定された配列あるいは代入された値を使ってプログラム display_2d が実行されます。

1.4.2 具体的な組み込み作業

以上の組み込み作業をもう少し具体的に示します。

- まず、プログラムの初めの部分 (int main(void) の前、#include..., #define... の後) に display_2d の記述を以下のように追加します。これが関数の定義になります。

($\mathbf{compo(G,s)}$ のプログラム全体の枠組み)

```
// Computing vertices connected to a specified one s in a graph
#include <stdio.h>
#define n 6
#define r 4
#define s 4

// 関数の定義ここから
void display_2d(const int M[][n+1], int p1, int p2, int q1, int q2, char b)
{
    int i, j;

    for (i=p1; i<=p2; i++){
        for (j=q1; j<=q2; j++){
            if (j==0){
                printf("%c[%d][%d]=%d: ", b, i, j, M[i][j]);
            }
            else if (M[i][j]==1){
                printf("%d ", j);
            }

        }
        printf("\n");
    }
}
// 関数の定義ここまで

int main(void)
{
    int i, j, k, sum;
    int A[n+1][n+1]={
        {0,0,0,0,0,0,0},{0,0,1,0,1,1,0},{0,1,0,1,0,1,0},{0,0,1,0,1,1,0},
        {0,1,0,1,0,0,0},{0,1,1,1,0,0,1},{0,0,0,0,0,1,0}
    };
    int N[r+1][n+1];
    int R[n+1];
      ...
      ...
 }
```

- 次に「隣接行列の表示」と「距離集合 N_k の表示」の部分の記述を以下のように変更します。比較のために、変更前の記述を /∗···∗/ で囲んでコメントとして残しています。これらがどちらも 1 行に置き換わることがはっきりと見えると思います。

「隣接行列の表示」部分の変更

```
// 隣接行列の表示
printf("\n***** 隣接行列（左端は各点の次数） *****\n");
display_2d(A,1,n,0,n,'A');

/*
for (i=1; i<=n; i++){
    for (j=0; j<=n; j++){
        if (j==0){
            printf("A[%d][%d]=%d: ",i,j,A[i][j]);
        }
        else if (A[i][j]==1){
                printf("%d ",j);
        }
    }
    printf("\n");
}
*/
```

「距離集合 N_k の表示」部分の変更

```
 // s からの距離が k>=0 の点の集合 N_0, ..., N_{r-1}, N_{r}の表示；
// N_{r}は空集合です
printf("\n***** %d からの距離が k>=0 の点の集合 N[0],...,N[%d](左端は要素
数) *****\n", s, r);
display_2d(N,0,r,0,n,'N');

/*
for (i=0; i<=r; i++){
    for (j=0; j<=n; j++){
        if (j==0){
            printf("N[%d][%d]=%d: ", i, j, N[i][j]);
        }
        else if (N[i][j]==1){
            printf("%d ", j);
        }
    }
    printf("\n");
}
*/
```

- これで、「隣接行列の表示」および「距離集合 N_k の表示」それぞれにおいて、display_2d の中の仮引数に対して受け取った具体的な配列や値を設定あるいは代入して (つまり実引数に対して) display_2d が実行されることになります。

1.4.3 ここまでのまとめ

皆さんに関数についての理解を深めていただくために、ここまでの関数についての説明を用語の解説も含めてまとめておきます。(関数については後で追加の説明をします。)

- プログラムの初めの部分 (`int main(void)` の前、`#include..., #define...` の後) に関数の記述を追加します。これが関数の定義になります。その際には、関数名の右側のカッコ内に仮引数のリストを書きます (各仮引数のデータ型も記載します)。
- 仮引数の並びに合わせて対応する実引数 (具体的な配列や数値) を記載します。これで、仮引数に対して配列を設定したり値を代入する、ということが行われます。これで関数に対して実行に必要なデータが渡されます。これを**データの受渡し** (data passing) といいます。(**引数の受渡し**という場合もあります。)
- 仮引数に実引数を渡すことで関数を実行させることを**関数の呼び出し** (function call) といいます。

繰り返しますと、実引数を仮引数に渡すこと (データの受渡し) は、関数名に続く仮引数の並びにおいて、渡すべき実引数を当該の仮引数の位置に記載することで行われます。プログラム中にこのような形で関数を記載することで実引数に対して関数が実行されます。これが関数の呼び出しです。

「隣接行列の表示」について言えば、以下の対応となります。

関数 `display_2d` の仮引数：	`M[][n+1], p1, p2, q1, q2, b`
実引数：	`A,1,n,0,n,'A'`
データの受け渡しと関数の呼び出し (実行)：	`display_2d(A,1,n,0,n,'A');`

注意 11.2

- 「隣接行列の表示」において、関数のデータの受渡しと呼び出し (実行) は

  ```
  display_2d(A,1,n,0,n,'A');
  ```

  となっています。仮引数 `M[][n+1]` に渡された実引数が `A` ということですが、変数の場合と違って**配列の場合には実引数は配列名のみを書く**ことになっています (この場合は `A` です)。このことは「距離集合 N_k の表示」でも同じで、2 次元配列でも 1 次元配列でも変わりません。
- 変数については、渡す実引数を書いています (実引数が仮引数に代入されます)。変数の場合にこの値を渡す方法を**値渡し** (pass by value) といいます。配列の場合は (配列内の多数のデータを渡すように思われるかもしれませんが) 値渡しではありません。現段階では残念ながらその説明は省略します。配列の場合は「実引数は配列名だけ書く」ということだけは記憶に留めておいてください。
- 例えば、`main` の中で `x=1; y=n;` と記述し、関数の呼び出しで `display_2d(A,x,y,0,n,'A')` と記述しても正常に動作します。`x, y` も実引数といいます。さらに、`p1=1; p2=n;` として `display_2d(A,p1,p2,0,n,'A')` と記述しても正常に動作します。実引数の `p1, p2` と仮引数の `p1, p2` は同じ文字列ですが、コンピュータ内では**仮引数と実引数は全く別物**として扱われます。一方の変化が他方に影響することはありませんが、同じ文字列は混同しやすいので注意が必要です。

注意 11.3 関数 `display_2d` を見ていただくと分かると思いますが、関数内で宣言した変数があります。この例では `int i, j;` です。関数は仮引数と関数内で宣言した変数を合わせ形で記述され、呼び出されると仮引数に実引数を受け取って (設定あるいは代入して) 関数の動作を実行します。

1.5 1 次元配列の表示について

n 次元のベクトルを 1 行 n 列の行列あるいは n 行 1 列の行列と考えることは数学や物理などではよくあることですが、コンピューターの配列の場合には 1 次元配列を 2 次元配列の 1 つとして扱うことは、現状の知識の範囲内では簡単なことではありません (できないことではないのですがいろいろと準備が必要となります)。プログラム compo.c の場合には 1 次元配列は表示部分の最後にある「点 s と連結な点の集合 R の表示」で扱うだけですので、関数を用意するメリットはほとんどありません。しかしながら、皆さんに 1 次元配列の関数での扱いを説明するためにここで取り上げることにしました。着目するのは以下の部分です。

```
103     // 点 s と連結な点の集合 R の表示
104     printf("\n***** 点 %d と連結な点の集合（左端は要素数） *****\n", s);
105     for (i=0; i<=n; i++){
106         if (i==0){
107             printf("R[%d]=%d: ", i, R[i]);
108         }
109         else if (R[i]==1){
110             printf("%d ", i);
111         }
112     }
113     printf("\n\n");
```

- ここで、以下の関数 display_1d を考えます。動作としては、整数型 1 次元配列 M[ ] の M[p] からの M[q] までの要素を横並びで表示させて最後に改行します。ただし、R[0] には R の要素数が入っていますので、$p = 0$ ならば R[0]=(**要素数**): という表示にしています。その右に続けて R[i]==1 なる点 i (R に含まれる点 i) を表示します。(このあたりの表現は前述の 2 次元配列の場合と同様です。)

```
void display_1d(const int M[], int p, int q, char b)
// displaying p-th element through q-th one of an 1d-array
{
    int i;

    for (i=p; i<=q; i++){
        if (i==0){
            printf("%c[%d]=%d: ", b, i, M[i]);
        }
        else if (M[i]==1){
            printf("%d ", i);
        }
    }
    printf("\n\n");
}
```

- この関数 display_1d の定義を display_2d の定義の次に置き、さらに以下のようにプログラムに記載して、引数の受渡しや呼び出し (実行) を行います。2 次元配列の場合と同様に変更前の記述を /∗···∗/ でコメントとして残しています。

```
// 点 s と連結な点の集合 R の表示
    printf("\n***** 点 %d と連結な点の集合（左端は要素数） *****\n", s);
```

```
   display_1d(R, 0, n, 'R');
/* for (i=0; i<=n; i++){
       if (i==0){
           printf("R[%d]=%d: ", i, R[i]);
       }
       else if (R[i]==1){
           printf("%d ", i);
       }
   }
   printf("\n\n"); */
```

1 次元配列の場合にもこのようにして関数を利用することができます。その他については 2 次元配列の場合と同様ですので、説明は省略します。

1.6 関数を組み込んだプログラム compo-func.c

ここまで説明してきた関数を組み込んだプログラム compo-func.c を図 121〜図 124 に示しておきます。なお、行番号を説明用に記載しています。実際のプログラムに記載するとエラーになります。

関数の利用で main の中で表示操作がいくらか見易くなったと思うのですが、皆さんはどのように感じておられるでしょうか。図 116 のグラフ G_4 に対する実行結果を図 125 示します (関数の組み込み前と同じです)。

```
1   // Computing vertices connected to a specified one s in a graph  compo-func.c
2
3   #include <stdio.h>
4   #define n 6
5   #define r 4
6   #define s 4
7
8   void display_2d(const int M[][n+1], int p1, int p2, int q1, int q2, char b)
9   // displaying rows from p1 through p2 and columns from q1 through q2
10  // of a 2d-array
11  {
12      int i, j;
13
14      for (i=p1; i<=p2; i++){
15          for (j=q1; j<=q2; j++){
16              if (j==0){
17                  printf("%c[%d][%d]=%d: ", b, i, j, M[i][j]);
18              }
19              else if (M[i][j]==1){
20                  printf("%d ", j);
21              }
22          }
23          printf("\n");
24      }
25  }
26
```

図 121　関数を組み込んだプログラム compo-func.c(その 1)

```
27  void display_1d(const int M[], int p, int q, char b)
28  // displaying p-th element through q-th one of a 1d-array
29  {
30      int i;
31
32      for (i=p; i<=q; i++){
33          if (i==0){
34              printf("%c[%d]=%d: ", b, i, M[i]);
35          }
36          else if (M[i]==1){
37              printf("%d ", i);
38          }
39      }
40      printf("\n\n");
41  }
42
43  int main(void)
44  {
45      int i, j, k, sum;
46      int A[n+1][n+1]={
47          {0,0,0,0,0,0,0},{0,0,1,0,1,1,0},{0,1,0,1,0,1,0},{0,0,1,0,1,1,0},
            {0,1,0,1,0,0,0},{0,1,1,1,0,0,1},{0,0,0,0,0,1,0}
48      };
49      int N[r+1][n+1];
50      int R[n+1];
51
52      printf("\n ***** Computing vertices connected to a specified one
                %d in a graph ******\n",s);
53
54      // 初期化
55      for (i=0; i<=r; i++){
56          for (j=0; j<=n; j++){
57              N[i][j]=0;
58          }
59      }
60      for (i=0; i<=n; i++){
61          R[i]=0;
62      }
63
64      // *** Step1 ***
65      k=1;
66
67      // *** Step2 ***
68      N[0][s]=1; N[0][0]=N[0][0]+1;
69      R[s]=1; R[0]=R[0]+1;
70
```

図 122　関数を組み込んだプログラム compo-func.c(その 2)

```
71      // *** Step3  拡大操作の反復 ***
72      // 3.1
73      while (N[k-1][0] > 0){
74
75          // 3.2 について：はじめに N[k][0..n] の要素はすべて 0 に初期設定されており、
76          //空集合への初期化は実行済みである
77
78          // 3.3 拡大操作
79          for (i=1; i<=n; i++){
80              if (N[k-1][i]==1){
81                  for (j=1; j<=n; j++){
82                      if ( (k==1 && A[i][j]==1) | ((k>1 && A[i][j]==1) &&
                                                    (R[j]==0 && N[k][j]==0)) ){
83                          N[k][j]=1;
84                          N[k][0]=N[k][0]+1;
85                      }
86                  }
87              }
88          }
89
90          // 3.4 N_k を R に追加；k を 1 だけ増加
91          if (N[k][0] > 0){
92              for (i=1; i <=n; i++){
93                  if (R[i]==0 && N[k][i]==1){
94                      R[i]=1;
95                      R[0]=R[0]+1;
96                  }
97              }
98          }
99          k=k+1;
100     }
101
102     // ***** 結果の表示 *****
103     // 点の次数の計算
104     for (i=1; i<=n; i++){
105         for (j=1; j<=n; j++){
106             A[i][0]+=A[i][j];//A[i][0]=A[i][0]+A[i][j];
107         }
108     }
109
110     // 隣接行列の表示
111     printf("\n***** 隣接行列（左端は各点の次数） *****\n");
112     display_2d(A,1,n,0,n,'A');
113
114     // s からの距離が k>=0 の点の集合 N_0, ..., N_{r-1}, N_{r}の表示；
115     // N_{r}は空集合です
116     printf("\n***** %d からの距離が k>=0 の点の集合 N[0],...,N[%d]
                                        （左端は要素数） *****\n", s, r);
117     display_2d(N,0,r,0,n,'N');
118
```

図 123　関数を組み込んだプログラム compo-func.c(その 3)

```
119     // 点 s と連結な点の集合 R の表示
120     printf("\n***** 点 %d と連結な点の集合（左端は要素数） *****\n", s);
121     display_1d(R, 0, n, 'R');
122
123     // **** Step4 ***
124     return 0;
125 }
```

図 124　関数を組み込んだプログラム compo-func.c(その 4)

```
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ gcc -o compo-func compo-func.c
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ ./compo-func

 ***** Computing vertices connected to a specified one 4 in a graph ******

***** 隣接行列（左端は各点の次数） *****
A[1][0]=3: 2 4 5
A[2][0]=3: 1 3 5
A[3][0]=3: 2 4 5
A[4][0]=2: 1 3
A[5][0]=4: 1 2 3 6
A[6][0]=1: 5

***** 4 からの距離が k>=0 の点の集合 N[0],...,N[4]（左端は要素数） *****
N[0][0]=1: 4
N[1][0]=2: 1 3
N[2][0]=2: 2 5
N[3][0]=1: 6
N[4][0]=0:

***** 点 4 と連結な点の集合（左端は要素数） *****
R[0]=6: 1 2 3 4 5 6
```

図 125　compo-func.c の実行結果

1.7　関数の一般形について

ここまでは、皆さんに説明する内容をシンプルにするために関数の機能としては少し限定した形で説明してきました。ここで関数の一般形について説明しておきます。

1.7.1　戻り値をもつ関数の例

次の関数の例を見てください。関数名は compsum としています。これは、1 次元配列 M[] の M[f] から M[t] までの要素の和を計算して、その結果 (整数) を呼び出し側に返す関数です。この返す値を**戻り値** (return value) と言います。

(1 次元配列 M[] の要素 M[f] から M[t] までの和を計算する関数 compsum)

```
int compsum(int M[], int f, int t)
```

```
{
    int i, s=0;

    for (i=f; i<=t; i++){
        s+=M[i];  //s=s+M[i];
    }

    return s;
}
```

いま #define m 12 とし、main では

```
int fm, tm, sum;
int A[m+1];
```

と宣言されているとして compsum の使い方について説明をします。

- 呼び出し側、例えば main から compsum を呼び出すには、main の中に

  ```
  sum=compsum(A, fm, tm);
  ```

  と記述します。ここで、fm, tm は $1 \leq fm, tm \leq 12$ なる具体的な整数値です。これで関数 compsum に配列 A[m+1] と 2 つの整数 fm, tm が渡されたことになります。[*5]
- compsum は 1 次元配列 A[m+1] の A[fm],$\cdots$,A[tm] の和を計算し、結果を変数 s に格納します。
- 最後に return s; で s の値を戻り値として呼び出し側の main に返します。これで main の整数変数 sum に s の値が入ります。sum=compsum(A, fm, tm); はこのことを意味しています。
- compsum(A, fm, tm) は文字列で書いていますが実体は戻り値 (和である整数値) です。int compsum(int M[], int f, int t) なる表現のはじめに int が書いてあるのは、戻り値が整数であることを示しています。

なお、A[fm],$\cdots$,A[tm] の和を計算してその結果を表示することは、例えば

```
sum=compsum(A, fm, tm);
printf("*** %d 月から %d 月までの合計は %d です。\n\n", fm, tm, sum);
```

といった記述になります。sum を使わないでこれを

```
printf("*** %d 月から %d 月までの合計は %d です。\n\n", fm, tm, compsum(A, fm, tm));
```

と記述しても同じことです。いずれでも皆さんが分かりやすい書き方を選べばいいと思います。分けて記述する方が戻り値の意味は分かりやすいかもしれません。

1.7.2 関数の呼び出しと戻り値

ここまでの例を使った説明である程度は理解していただいたことと思いますが、まとめの意味で少し一般的な説明をしておきます。

- 関数の一般形は

[*5] 厳密には、配列については A[m+1] のメモリ内での存在場所を知らせているのですが、ここでは詳細は省いて、配列を渡したことになると考えてください。変数については、f ← fm、t ← tm と実引数が仮引数に代入されます。

戻り値のデータ型　関数名（仮引数 1 のデータ型 仮引数 1, ... ）
```
{
   ......
   return（戻り値を格納する）変数名;
}
```
です。

- すなわち、関数名の右側のカッコ内にデータ型付きの仮引数のリストを、関数名の左側に戻り値のデータ型を置き、記述の最後に return **変数名** の形で戻り値を格納する変数を return 文に記載します。戻り値を格納する変数は関数内の宣言に記載し、関数内で値を設定します。
- この関数を呼び出してそのあとで実行結果を受け取るには、呼び出し側で (戻り値と同じデータ型の) 適当な変数を用意して、以下の形で関数名とカッコ内に実引数のリストを置きます。

  戻り値を受ける変数 = 関数名（実引数 1, ...);

  ただし、渡す実引数については、この実引数はこの仮引数に、···、と仮引数の順序に揃えて記載する必要があります。なお、戻り値を受ける変数を用意しないで、**関数名（ 実引数 1, ··· ）**を直接戻り値として使用することも可能です。
- 関数を組み込んだプログラムの例として示した compo-func.c では以下の関数を使いました。

```
void display_2d(const int M[][n+1], int p1, int p2, int q1, int q2, char b)
void display_1d(const int M[], int p, int q, char b)
```

  いずれも戻り値のデータ型が void となっています。これは戻り値がない (あるいは何も返さない) 場合の関数の記述です。void は「空 (から)」の意味を持っており、何もないものを返す、ということで関数の形に合わせた記述になっています。

注意 11.4 ここまでの説明で main も関数の形をしていることに気が付かれたでしょうか。

```
int main(void)
```

という形をしています。void と記載ですので仮引数はありません。戻り値は最後の return 0; によって返される整数 0 です。戻り値はどこに返されているのでしょうか。とりあえずは、

> OS が戻り値を受け取って、0 ならばプログラムは正常に終了したと判断し、それ以外を受け取ればエラーとみなす

と考えてください。return 文の実行でプログラムは終了ですが、main の最後に return 文が無い場合もあります。その時は最後の } を読んだ時点で終了します。

1.7.3 戻り値を持つ関数 compsum の使用例

戻り値を持つ関数 compsum を使用するプログラムの例を図 126、図 127 に示します。以下の動作をします。

1. #define m 12 として、整数型 1 次元配列 A[m+1] の A[1] ··· A[m] に 1 月から m 月までそれぞれの月毎に 1 つの整数値が格納されているとします。例えば、毎月の車の販売台数などのイメージです。
2. まず、配列 A の A[1] から A[m] までの要素を A[m]=(**値**) の形で横並びに表示させます。
3. 次に、$1 \leq m1 \leq m2 \leq n$ なる 2 つの整数 $m1, m2$ をキーボードから入力させてこれらをそれぞれ整数型変数 fm, tm に格納します。

4. その後、main から fm, tm を実引数として関数 compsum を呼び出して、fm 月から tm 月までの整数値の合計を計算させます。
5. compsum は計算終了後に合計を呼び出し側 main に戻り値として返します。main では戻り値を整数型変数 sum に受け取って値を表示します。
6. ただし、$m1 < 0$ あるいは $m2 < 0$ なる負の整数が入力された時にはすぐにプログラムを終了させます。また、$1 \leq m1, m2 \leq n$ であっても、$m1 > m2$ のときはその旨を表示してやはりすぐにプログラムを終了させます。$m1, m2$ の入力時に以上のことを判定しながら操作を進めていきます。

```
1  // Program example with return value
2
3  #include <stdio.h>
4  #define m 12
5
6  int compsum(int M[], int f, int t)
7  {
8      int i, s=0;
9
10     for (i=f; i<=t; i++){
11         s+=M[i];
12     }
13
14    return s;
15 }
16
17 void hyouji(const int M[], int p, int q, char b)
18 // 1 次元配列の要素 M[p]〜M[q] の表示
19 // 表示は b[i]=値 ... の形：b は指定した文字
20 {
21     int i;
22
23     for (i=p; i<=q; i++){
24         printf("%c[%d]=%d ", b, i, M[i]);
25     }
26     printf("\n\n");
27 }
28
```

図 126　関数 compsum を組み込んだプログラム例 prog-compsum.c (その 1)

- プログラム名は prog-compsum.c としています。なお、掲載のプログラム中では、1 次元配列 A のデータはすべて 10 としています。これはプログラムが正常に動作していることの確認を容易にするためで、特に意味はありません。皆さんがこのプログラムを実行する機会があれば、是非いろいろな数値データに対して実行させてみてください。
- 図 128 に、実際にいくつかの数値を入力した場合のプログラムの実行結果を示しています。上述した $m1 < 0$ あるいは $m2 < 0$ なる負の整数入力や、$1 \leq m1, m2 \leq n$ であっても $m1 > m2$ なる入力に対してすぐに終了しているのが分かると思います。

```
29 int main(void)
30 {
31     int fm=1, tm=1, sum;
32     int A[m+1]={0,10,10,10,10,10,10,10,10,10,10,10,10};
33
34     printf("***** 指定期間内の数値の合計計算 *****\n\n");
35     printf("***** 1 月から 12 月までのデータ *****\n");
36     hyouji(A,1,12,'A');
37     while (fm > 0 && tm > 0){
38         printf("*** 開始の月を入力してください（負の整数を入力すると終了します）：");
39         scanf("%d",&fm);
40         if (fm > 0){
41             printf("*** 終了の月を入力してください（負の整数を入力すると終了しま
す）：");
42             scanf("%d",&tm);
43             if  (tm > 0){
44                 if (fm <= tm){
45                     sum=compsum(A, fm, tm);
46                     printf("*** %d 月から %d 月までの合計は %d です。
\n\n", fm, tm, sum);
47                 }
48                 else{
49                     printf("開始が %d 月で終了が %d 月ですので終了します。再実行してく
ださい。\n", fm, tm);
50                     fm=-1;
51                 }
52             }
53         }
54     }
55     printf("終了\n\n");
56
57     return 0;
58 }
```

図 127　関数 compsum を組み込んだプログラム例 prog-compsum.c (その 2)

- 実行結果の 1 行目はソースプログラム prog-compsum.c のコンパイル (およびリンク) の指示であり、2 行目が (実行可能プログラム prog-compsum に対する) 実行の指示です。実行可能プログラム名は (.c を除くだけで) ソースプログラムと同じにしています。途中で終了した場合には実行可能プログラムに対する実行の指示があります。

1.7.4　プログラム内での関数の記載位置について

ここまでの説明では、すべての関数は #include <stdio.h>[*6]や #define m 12[*7]などの記述と main の間の位置に置いています。すべての関数を前に置き、最後に main を置く形です。

[*6] ファイル名の後ろが.h の形のファイルは**ヘッダ** (header) と呼ばれます。stdio.h は C 言語の開発環境に標準で用意されているファイルです。プログラムの実行には必須のファイルですので、このヘッダーファイルをまず読み込みなさい、という指示です。

[*7] マクロの定義です。ここでの m を**マクロ名** (macro name) といいます。#が付いた記述をマクロの定義といい、この定義ではプログラムの中に出現する m はすべて値 6 に置き換えられてコンパイルされます。

```
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ gcc -o prog-compsum prog-compsum.c
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ ./prog-compsum

***** 指定期間内の数値の合計計算 ********** 1 月から 12 月までのデータ *****
A[1]=10 A[2]=10 A[3]=10 A[4]=10 A[5]=10 A[6]=10 A[7]=10 A[8]=10 A[9]=10 A[10]=10
A[11]=10 A[12]=10

*** 開始の月を入力してください (負の整数を入力すると終了します):1
*** 終了の月を入力してください (負の整数を入力すると終了します):12
*** 1 月から 12 月までの合計は 120 です。

*** 開始の月を入力してください (負の整数を入力すると終了します):3
*** 終了の月を入力してください (負の整数を入力すると終了します):9
*** 3 月から 9 月までの合計は 70 です。

*** 開始の月を入力してください (負の整数を入力すると終了します):-1
終了

Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ ./prog-compsum

***** 指定期間内の数値の合計計算 ********** 1 月から 12 月までのデータ *****
A[1]=10 A[2]=10 A[3]=10 A[4]=10 A[5]=10 A[6]=10 A[7]=10 A[8]=10 A[9]=10 A[10]=10
A[11]=10 A[12]=10

*** 開始の月を入力してください (負の整数を入力すると終了します):5
*** 終了の月を入力してください (負の整数を入力すると終了します):-1
終了

Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ ./prog-compsum

***** 指定期間内の数値の合計計算 ********** 1 月から 12 月までのデータ *****
A[1]=10 A[2]=10 A[3]=10 A[4]=10 A[5]=10 A[6]=10 A[7]=10 A[8]=10 A[9]=10 A[10]=10
A[11]=10 A[12]=10

*** 開始の月を入力してください (負の整数を入力すると終了します):8
*** 終了の月を入力してください (負の整数を入力すると終了します):5
開始が 8 月で終了が 5 月ですので終了します。再実行してください。
終了
```

図 128　プログラム prog-compsum.c の実行結果

- それでは、関数の個数が増えてきたり、長い記述の関数が出てきた場合を想像してみてください。プログラムの前にたくさん関数が並んでいて、はるか後ろの方に短い main がある状況です。main は関数の並びなどを含めて全体の流れを表していますので、この形のプログラムは決して分かり易いとは言えません。
- 逆に、main を先頭にしてその後ろに関数を置く形は、まず全体が見えてから詳細 (関数) が後ろに続きますので、分かり易くなるだろう、と予想できると思います。
- 実際このような記述方法はあります。ただし、単純に前後の置き換えをしただけでは実行されません。

関数原型宣言あるいは**プロトタイプ宣言** (function prototype declaration) と呼ばれる記述が必要になりますが、残念ながらここではその説明はしないことにします。いろいろと準備が必要になるからです。今後に向けた 1 つのキーワードとして用語だけを紹介しました。

1.8 まとめ

第 10 回で、アルゴリズム **compo(G, s)** をプログラム `compo.c` として実現しました。その際には正しく動くプログラムを作ることに重点を置きました。今回は、プログラムをより洗練されて読みやすく、改良や拡張などの今後の扱いが容易な形に修正することに挑戦しました。具体的には、新たに関数という仕組みを導入し、関数を取り込むことでプログラムの流れをクリアに理解しやすい形にすることです。

プログラム `compo.c` の作成では 2 次元配列や 1 次元配列の要素を表示する操作を追加しました。今回は、2 次元配列 `A` と `N` の要素表示の操作に対して、1 つの 2 次元配列の指定した範囲にある要素を表示する関数を用意しました。そして `A` と `N` それぞれについて、配列自体とその表示範囲や表示文字などの必要データを関数に与えることで 2 つの配列の表示操作を実現する、という関数の使用方法を示しました。

ここまで扱った関数はいずれも戻り値のない場合でした。ただし、戻り値を持つのが関数の一般形ですのでそのような関数の利用を含むプログラム作成の過程も示しました。具体的には、数値データを持つ 1 次元配列の指定した範囲にある数値の総和を求める関数を用意しました。動作としては、まず `main` 側で処理対象とする 1 次元配列の全要素を表示し、和を計算する範囲を入力させます。次に、これらのデータを関数に渡して範囲内の数値の総和計算を実行させ、最後にその計算結果を `main` で受け取って表示する、というプログラムを作成しました。

あまり専門的な事項に深入りせず、これまで説明してきた事項を使ってできる範囲内で関数の利用によるプログラム作成を説明しました。今回の説明が皆さんにとって関数の利用に慣れることの手助けになれば幸いです。