# プログラミング言語
# 演習IV

2023年７月21日

前橋工科大学　生命情報学科
中村 建介

# アルゴリズム

数値計算

サーチ、ソート

ツリー構造・グラフ

再帰

まず素直なコードを書く

それでも遅いとき　→　対策を考える

## システムに依存

1. コンパイルオプション(-O3)
2. 組み込み関数を使う

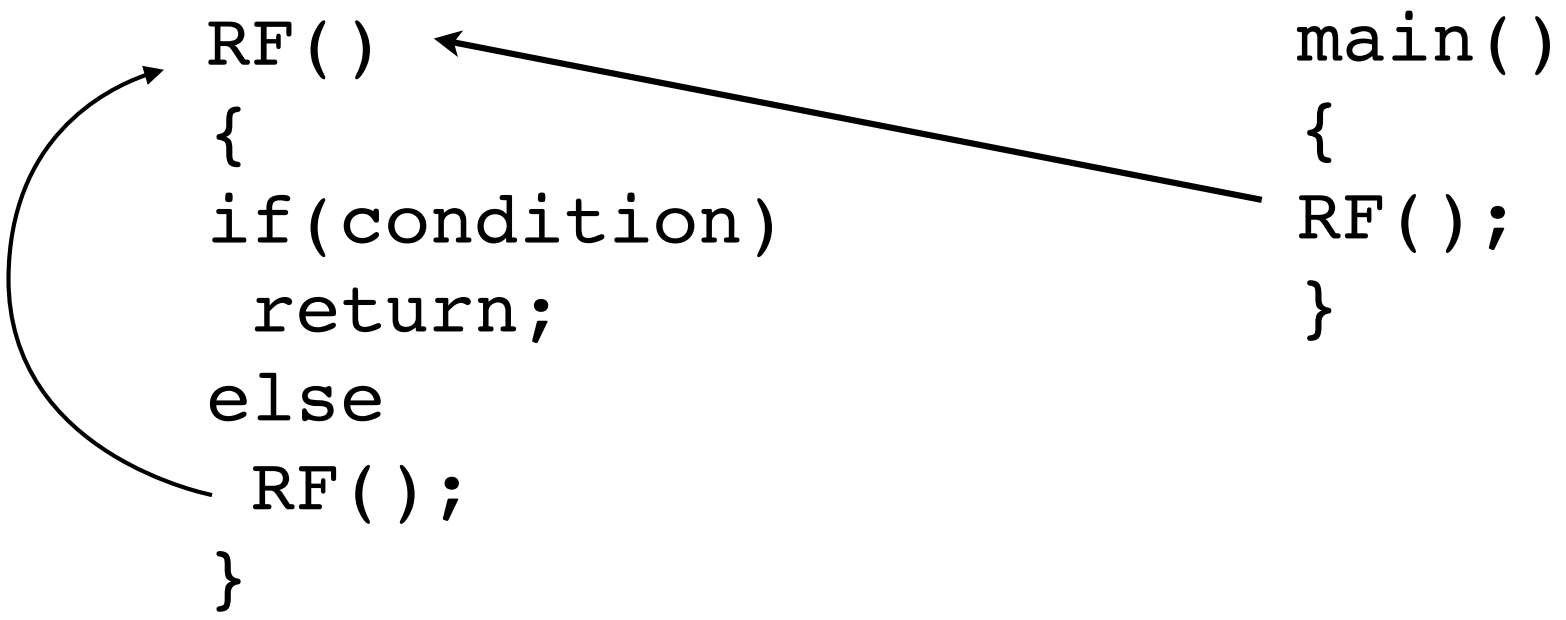
## アルゴリズムの工夫

1. 計算量を減らす
2. 計算機の気持ちになる
3. 全く違う発想・視点
4. 並列化

# 再帰関数

Recursive

自分自身を呼び出す関数



# 階乗

$$n! = \begin{cases} 1 & (n=0\text{のとき}) \\ n \times (n-1)! & (\text{それ以外}) \end{cases}$$

```
long  kaijo(long n)
{
if(n==0)
 return (1);
else
 return(n*kaijo(n-1));
}
```

$$4! = 4 \times 3 \times 2 \times 1$$

$$n! = \begin{cases} 1 & (n=0\text{のとき}) \\ n \times (n-1)! & (\text{それ以外}) \end{cases}$$

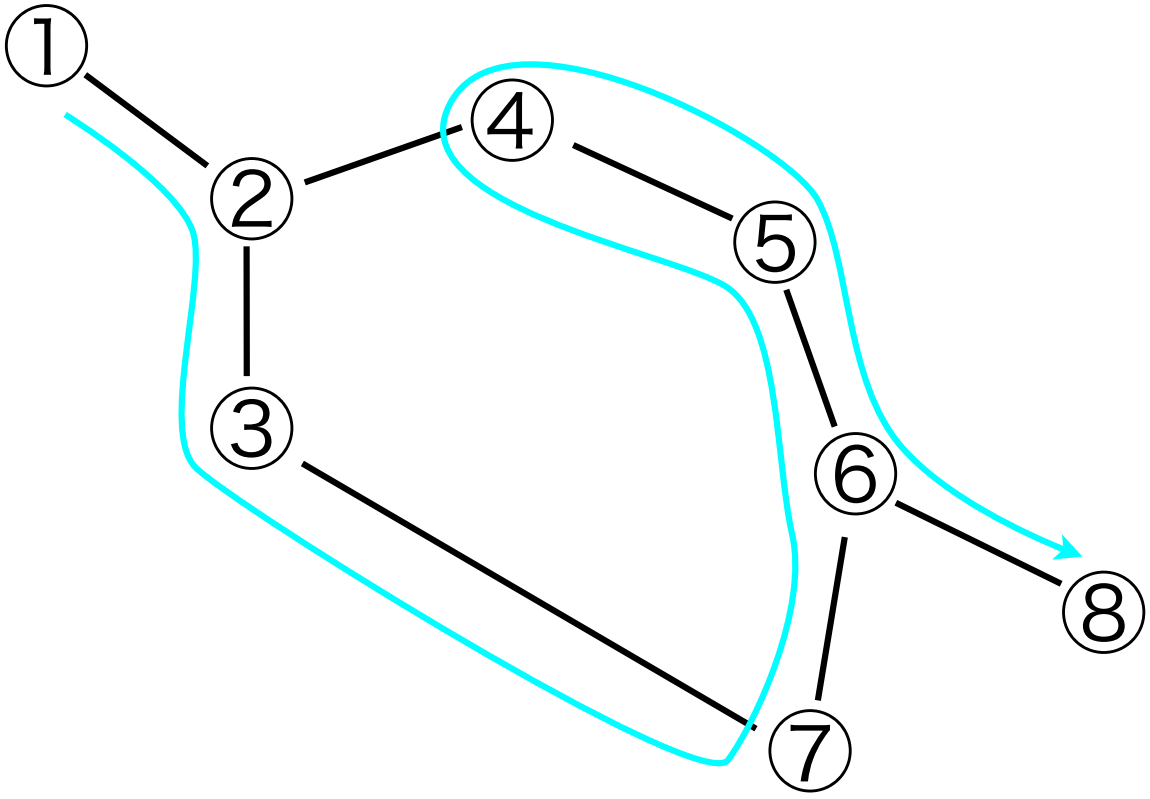
$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times 3 \times 2! \\ &= 4 \times 3 \times 2 \times 1! \\ &= 4 \times 3 \times 2 \times 1 \times 0! \\ &= 4 \times 3 \times 2 \times 1 \times 1 \end{aligned}$$

# Ackerman関数

$$
\mathtt{Ack}(m,n) = \begin{cases} n+1 & (m=0\text{のとき}) \\ \mathtt{Ack}(m-1,1) & (n=0\text{のとき}) \\ \mathtt{Ack}(m-1,\mathtt{Ack}(m,n-1)) & (\text{それ以外}) \end{cases}
$$

m, n の増加につれて爆発的に大きな数値になる

# グラフの探索




隣接行列

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 7 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

# 深さ優先探索

出発点を決めて隣接するノードの数値の若いものの順番に繋げて行く

①からは②へしか繋がっていないので②へ

②から①は既に通っているのでまず④より若い③へ

③から②は既に通っているので⑦へ

⑦から③は既に通っているので⑥へ

⑦から③は既に通っているので⑥へ

⑥から⑧より若い⑤へ

⑤から④へ

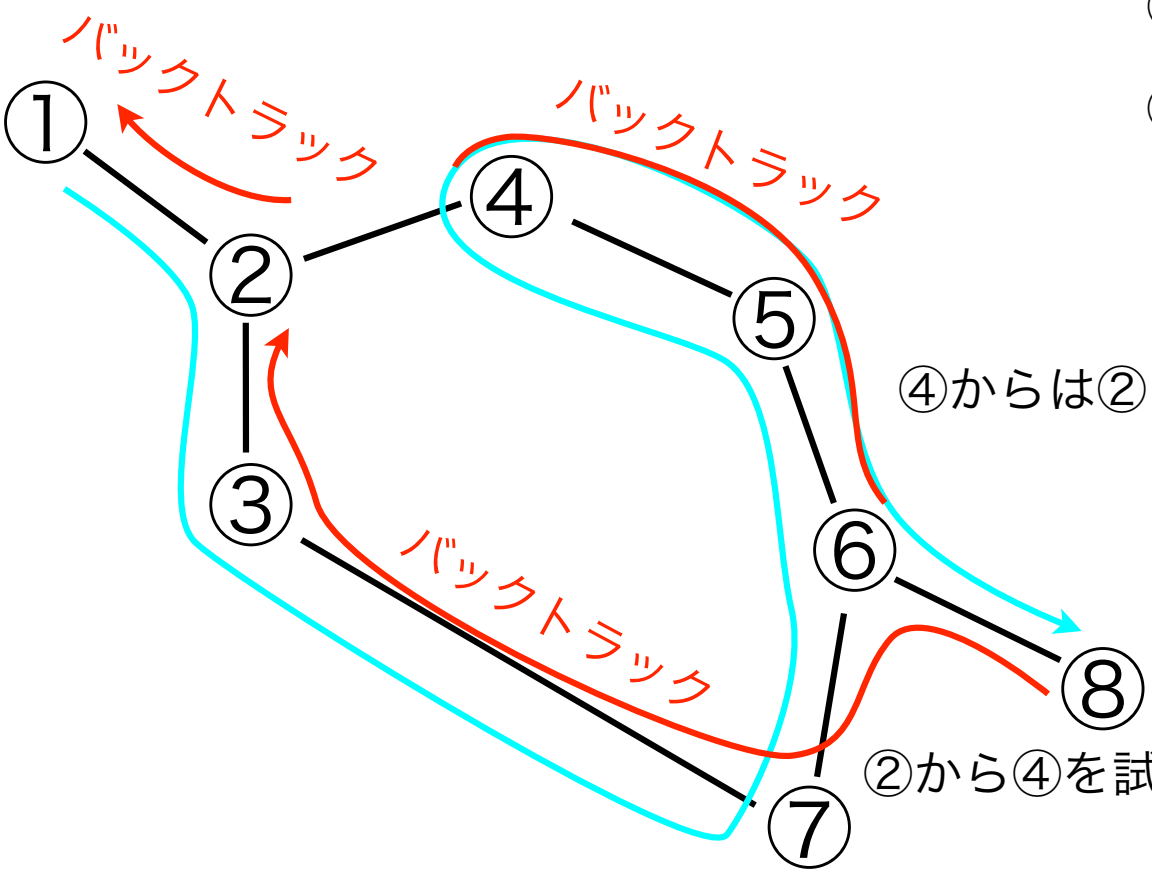
④からは②も⑤も既に通っているので行き止まり

⑥まで戻るバックトラック

⑥からまだ行っていない⑧へ

②まで戻るバックトラック

②から④を試すが既に通っているので①まで戻る



# 課題１

ある場所で①から⑧について走査

繋がっている場合、移動を検討する

既に通っている（訪問フラグ＝１）場合には移動しない

移動した場合にはまずその場所の訪問フラグを１にする

隣接行列a[ ][ ]：繋がっているノード間は１

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 7 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

訪問フラグ v[ ]：既に通った場所は１にする

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

```
void visit(int now)
{
int next;
v[now] = 1;
for(next=0;next<N;next++)
{
if((a[now][next] == 1) && (v[next] == 0))
 {
 printf("%d->%d, ",now,next);
 visit(next);
 }
}
```

今来た場所を訪問済みにする

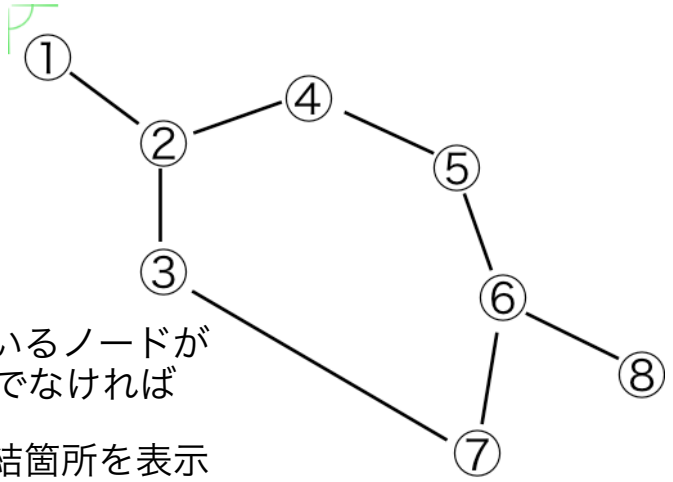
N: ノード数＝８

今いるノードと行こうとしているノードが繋がっていれば

行こうとしているノードがまだ訪問済みでなければ

見つけた新たな連結箇所を表示

次のノードを添字にして**再帰呼び出し**



main関数で隣接行列・訪問フラグ等を定義した上で任意のノード名でvisit関数を呼び出す

①からなら②が先ず繋がっているので③以降のチェックは後回しにして（深さ優先）③へ移動

③から②へは訪問フラグが立っているので行かず、⑦へ

同様に⑦からは⑥へ⑥からは⑧は後回しで⑤へ⑤から④

④からは②へ繋がっているが訪問済みなので⑤に戻る

⑤から繋がっているノードは調査済/訪問済なので⑥に戻る

⑥からは⑧への訪問が未チェックで可能なので⑧へ行く

⑧からは行き先がないので⑥へ戻る

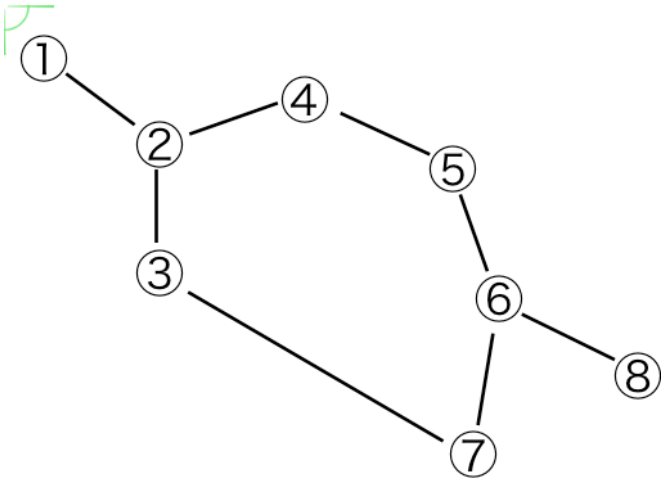
②まで戻り④が訪問済みなのをチェックして①まで戻ってmainに帰る

隣接行列a[ ][ ]：繋がっているノード間は1

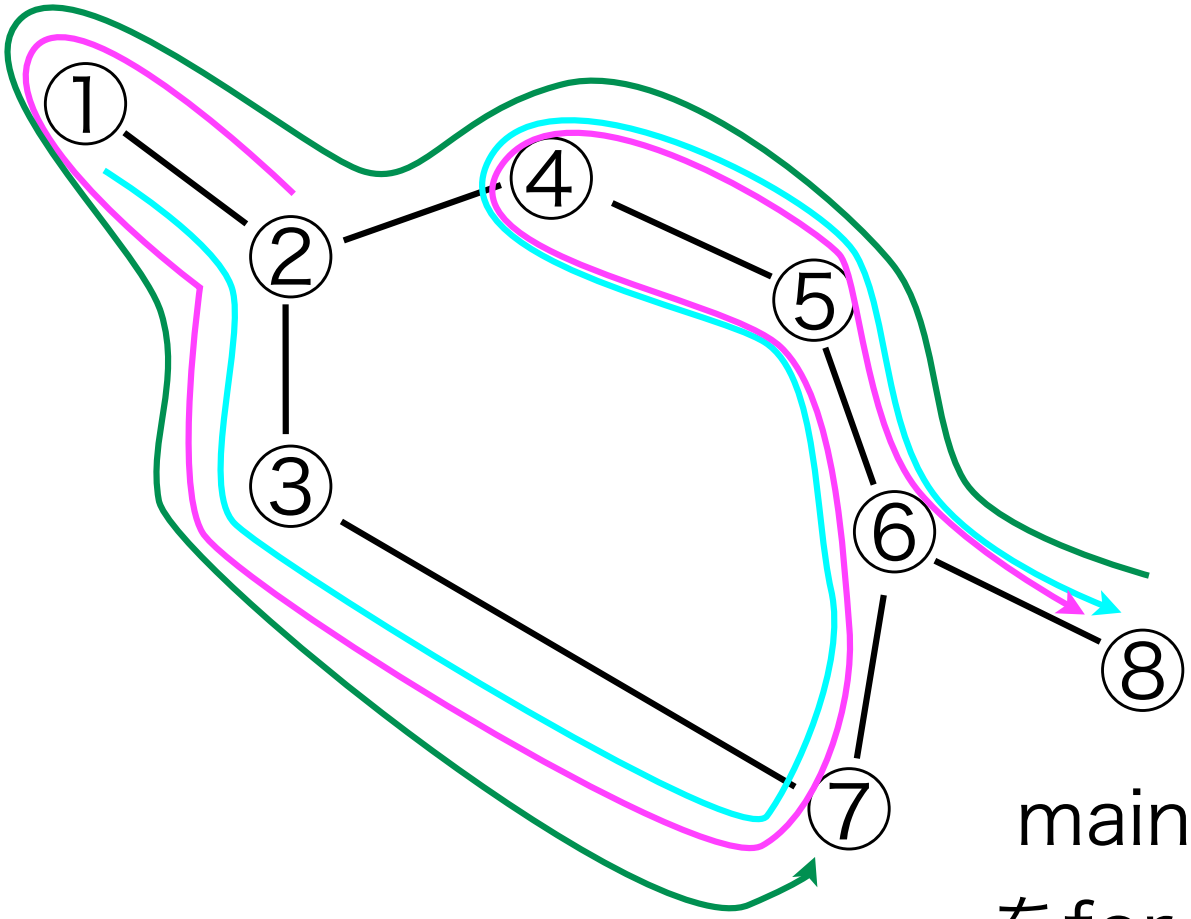
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 7 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

訪問フラグ v[ ]：既に通った場所は1にする

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



# 網羅的探索 課題２




①を起点とした場合

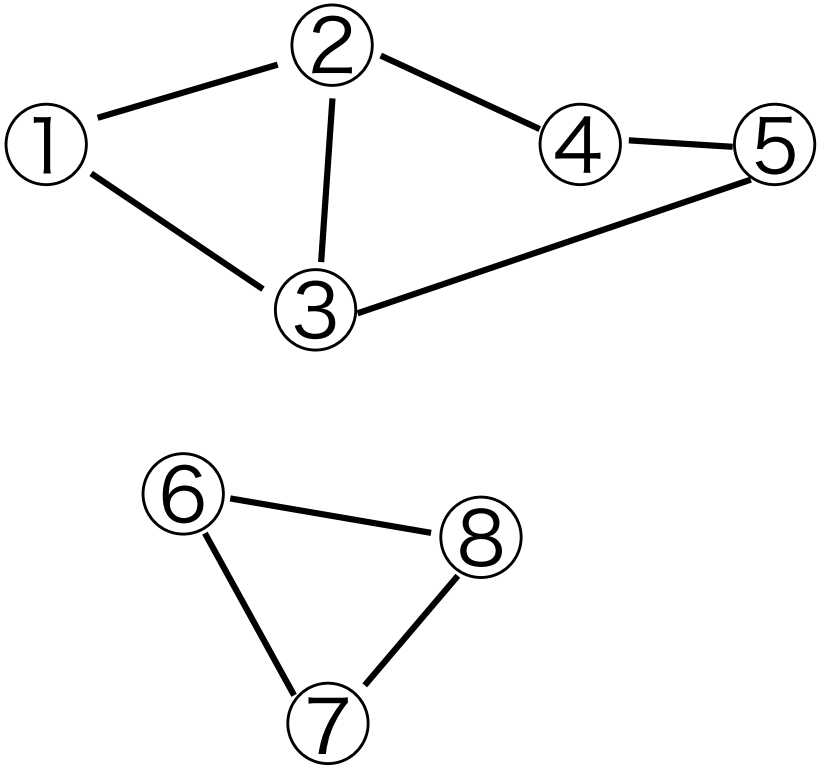
②を起点とした場合

.....

⑧を起点とした場合

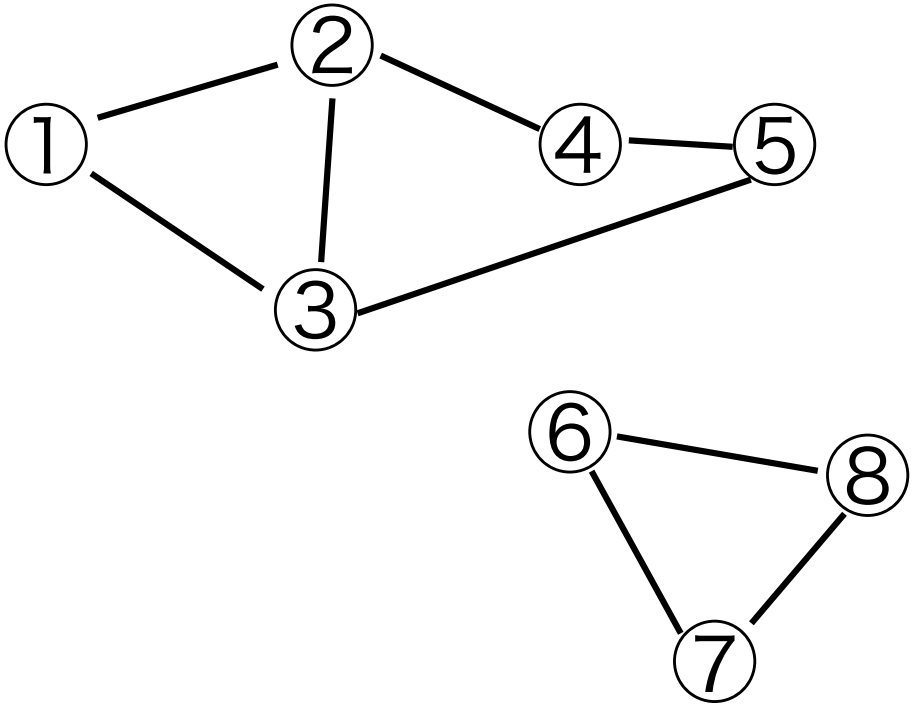
mainから呼び出すvisitの添字をforループで1-8に振ればよい

# 非連結グラフ　課題３




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

# 非連結グラフ　課題３

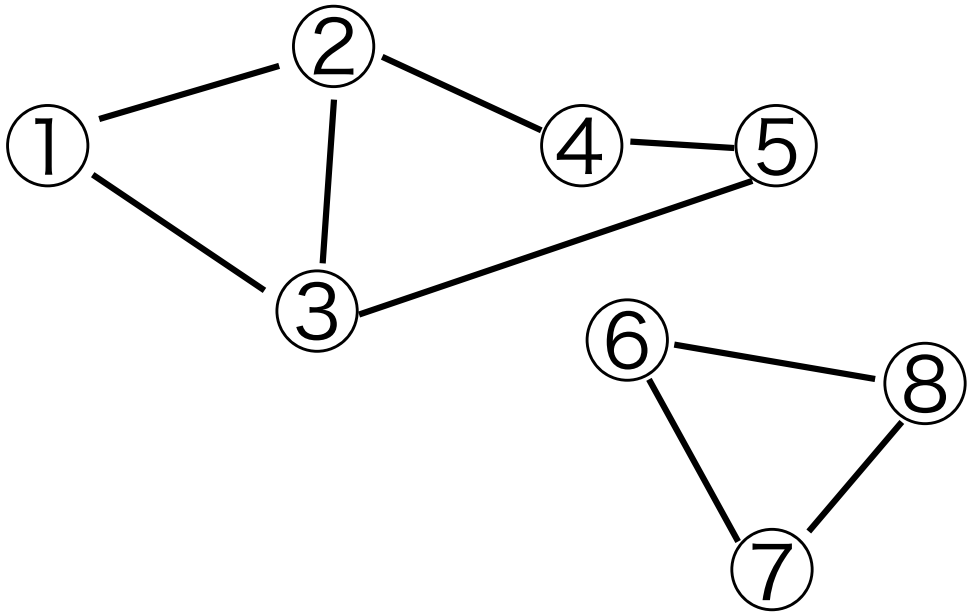



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

出力

グラフ１：1, 2, 3, 4, 5

グラフ２：6, 7, 8




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

一つ目のグラフ{①,②,③,④,⑤}

二つ目のグラフ{⑥,⑦,⑧}の要素を列挙

mainの中から、全ての未訪問のノードを起点として

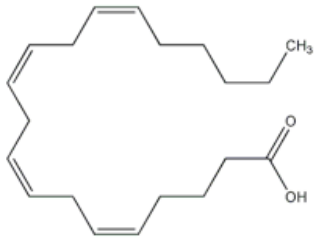
visitを試みる

```
int graph_num = 1;  //独立したグラフの数
for(i=1;i<=8;i++)
 {
 if(v[i] == 0)
  {
  printf(“Graph %d:”,graph_num ++);
  visit(i);
  printf(“\n”);
  }
 }
```

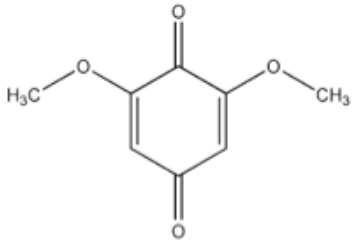
```
void visit(int now)
{
int next;
v[now] = 1;
for(next=0;next<N;next++)
{
if((a[now][next] == 1) && (v[next] == 0))
 {
 visit(next);
 }
}
```

# 化学構造

c_1_0

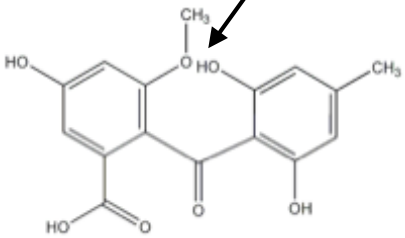


c_2_1

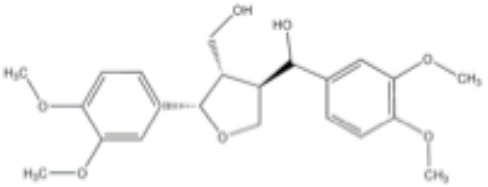


c_3_2

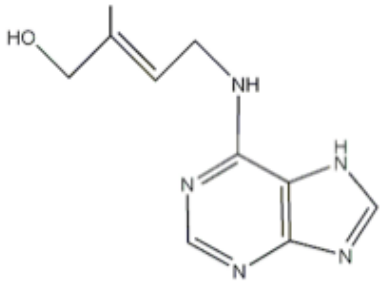
閉じてない



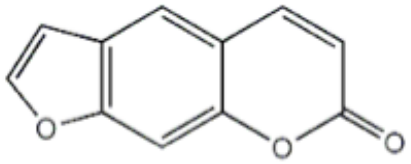
c_4_3



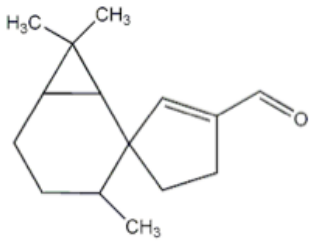
c_5_2



c_6_3



c_7_3

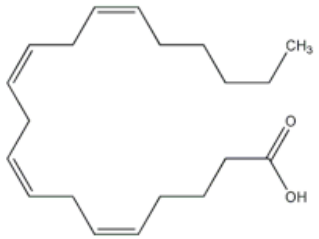


# 環を認識する

# 課題4

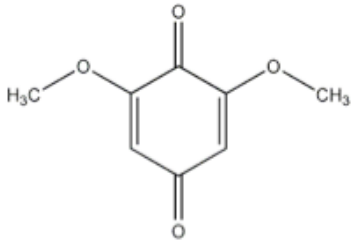
C_PROG_14_04

c_1_0



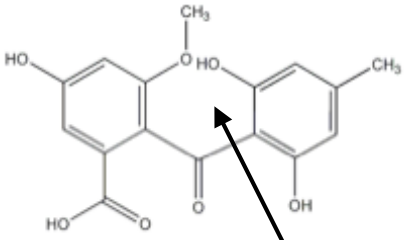
0

c_2_1



1

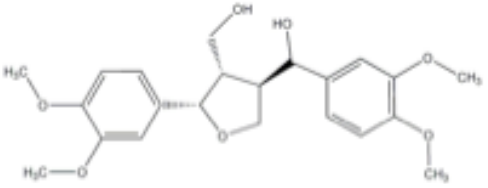
c_3_2



2

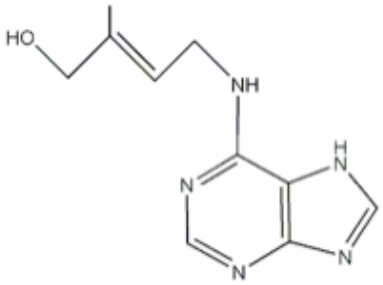
閉じてない

c_4_3



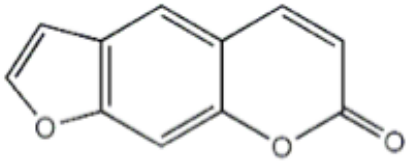
3

c_5_2



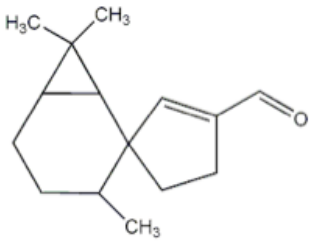
2

c_6_3



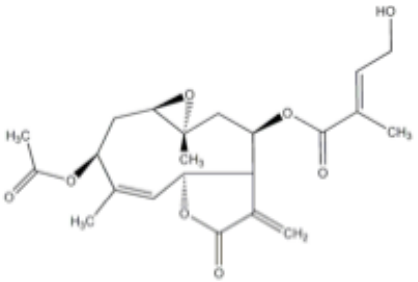
3

c_7_3



3

c_8_3



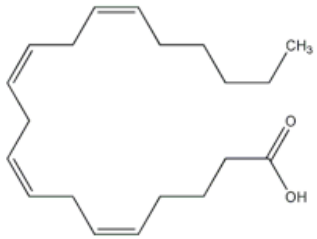
3

環の数をカウントする

# 応用課題5

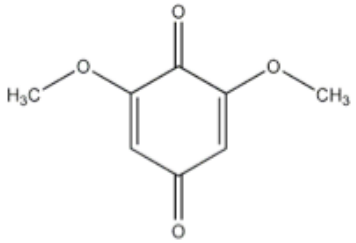
C_PROG_14_05

c_1_0



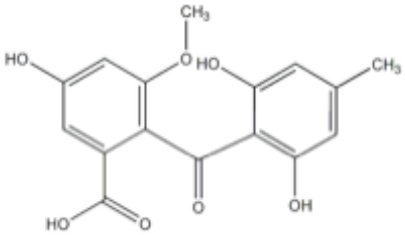
0

c_2_1



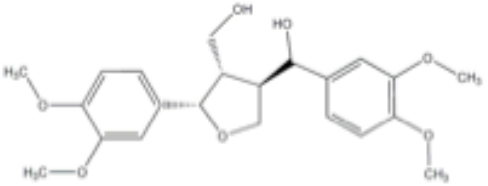
6 - 1

c_3_2



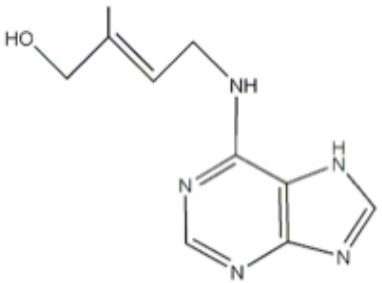
6 - 2

c_4_3



5 - 1
6 - 2

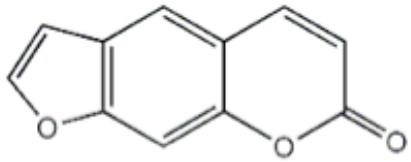
c_5_2



**5 - 1**
**6 - 1**

9 - 1

c_6_3



**5 - 1**
**6 - 2**

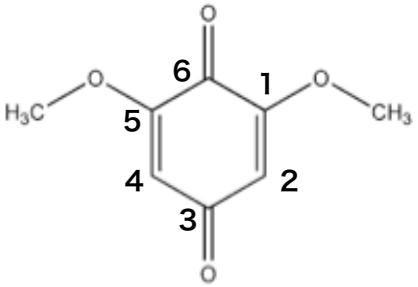
9 - 1
10 - 1
13 - 1

環の員数をカウントする

# 応用課題6

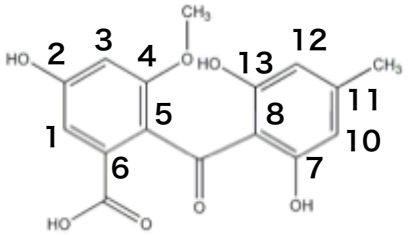
C_PROG_14_06

c_2_1




ring1 : 1-2-3-4-5-6

c_3_2




ring1 : 1-2-3-4-5-6

ring2 : 7-8-13-12-11-10

環を構成する原子をリストアップする

# mol 形式のファイルを読み込む

８ 回目　6/9　のプログラムを流用

# 原子間の隣接行列を作る

先週はハードコードしてあった、
２次元配列をmallocまたは決めうち

# 隣接行列をトレースして環の部分を検出する

先週のプログラム：再帰による深さ優先探索を応用

隣接行列

二次元配列

最大の原子数で決めうち

```
connect[MAX_ATOM][MAX_ATOM];
```

MAX_ATOM = 100 程度

ポインタの配列をとって malloc

```
int **connect;
```

# 環のカウント

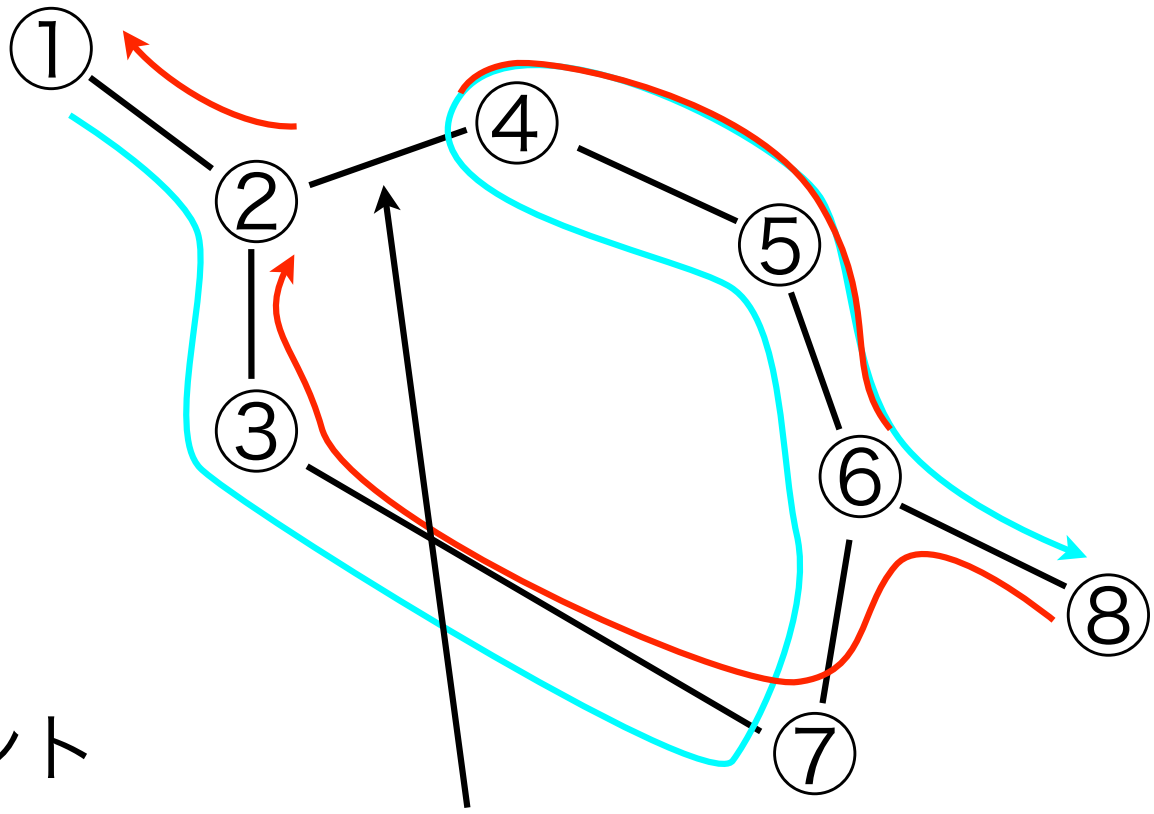
```
int n_ring = 0;
```

深さ優先探索で

④から②を試す時に

自分以外で訪問フラグ
が立っている場合

環の数をひとつとカウント




自分以外で訪問フラッグが立ってる
場所にぶつかった時

```
n_ring ++;
```