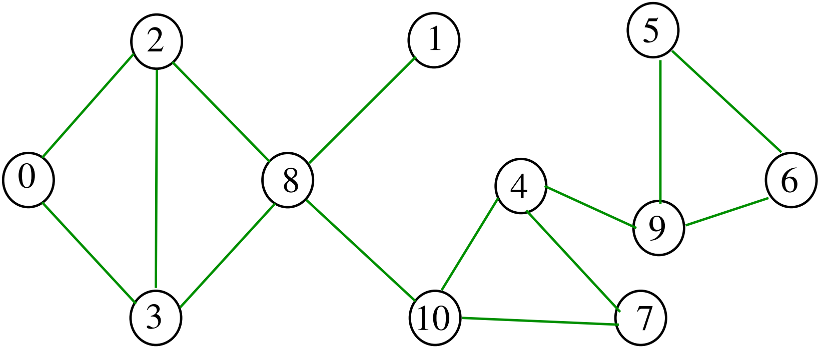
Melhores momentos

AULA 10

Pontes em grafos

Uma aresta de um grafo é uma **ponte** (= *bridge* = *separation edge*) se ela é a única aresta que atravessa algum corte do grafo.

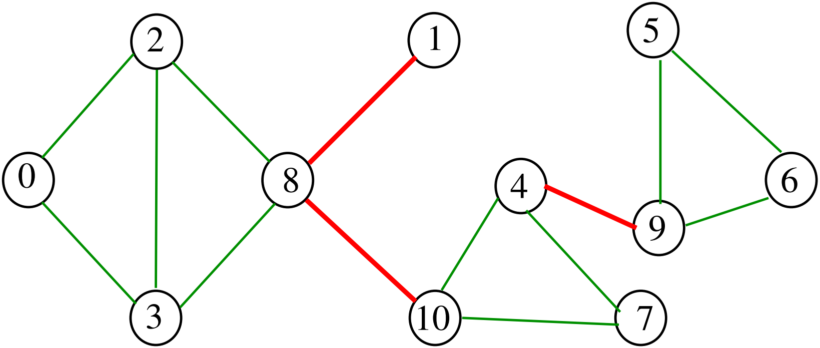
Exemplo:



Pontes em grafos

Uma aresta de um grafo é uma **ponte** (= *bridge* = *separation edge*) se ela é a única aresta que atravessa algum corte do grafo.

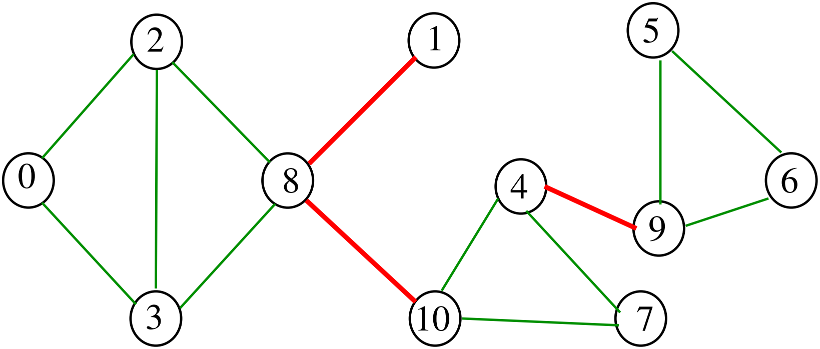
Exemplo: as arestas em vermelho são pontes



Procurando pontes

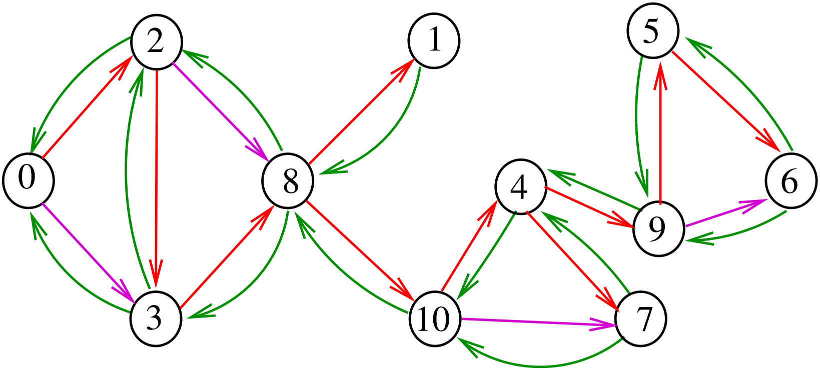
Problema: encontrar as pontes de um grafo dado

Exemplo: as arestas em vermelho são pontes



Propriedade

Um arco v-w da floresta DFS faz parte (juntamente com w-v) de uma ponte se e somente se não existe arco de retorno que ligue um descendente de w a um ancestral de v



Aresta-biconexão

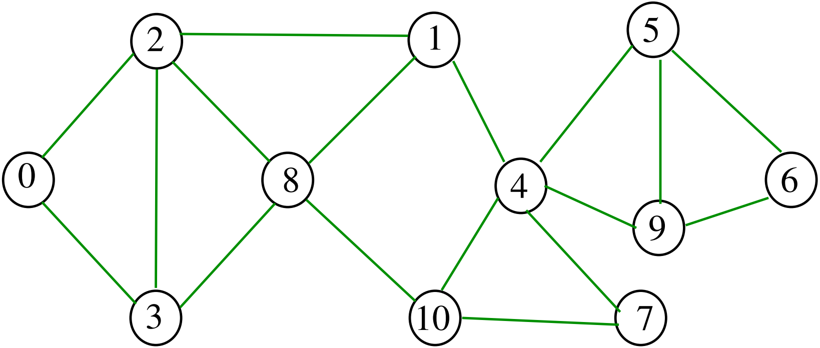
Um grafo é **aresta-biconexo** (= `2-edge-connected`) ou **2-aresta-conexo** se for conexo e não tiver pontes.

Fato básico importante:

> Um grafo é aresta-biconexo se e somente se, para cada par (`s`,`t`) de seus vértices, existem (pelo menos) dois caminhos de `s` a `t` sem arestas em comum.

Exemplo

É preciso remover pelo menos duas arestas de um grafo aresta-biconexo para que ele deixe de ser conexo



AULA 11

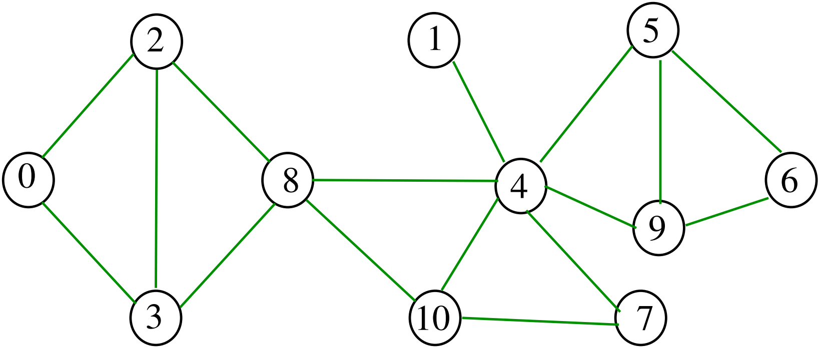
Articulações e biconexão

S 18.6

Articulações em grafos

Uma **articulação** (= *articulation point*) ou **vértice de corte** (= *cut vertex*) de um grafo é um vértice cuja remoção aumenta o número de componentes

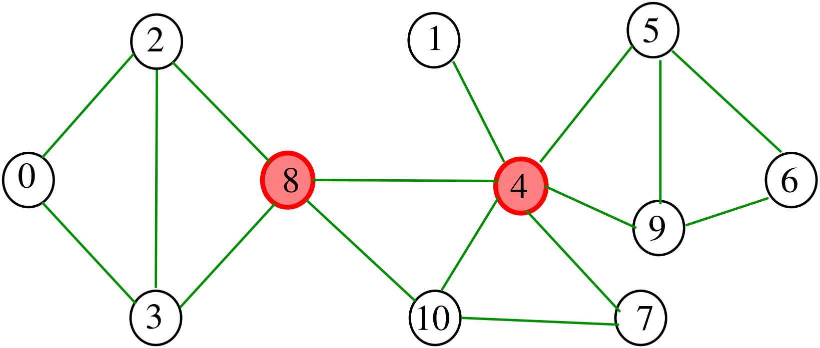
Exemplo:



Articulações em grafos

Uma **articulação** (= *articulation point*) ou **vértice de corte** (= *cut vertex*) de um grafo é um vértice cuja remoção aumenta o número de componentes

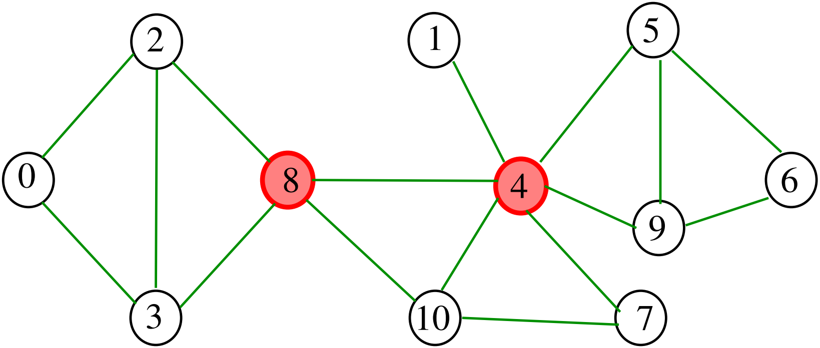
Exemplo: os vértices em vermelho são articulações



Procurando articulações

Problema: encontrar as articulações de um grafo

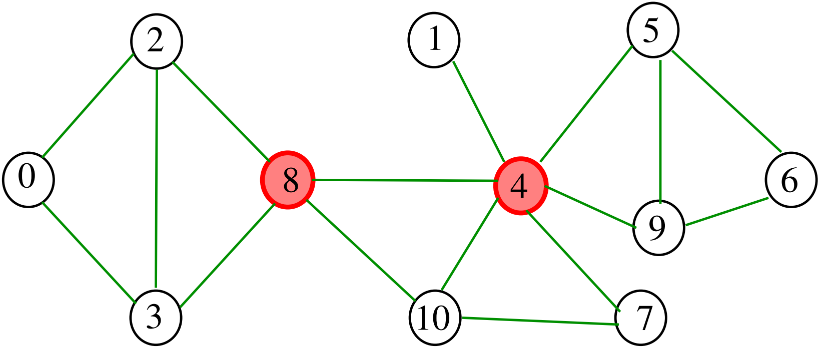
Exemplo: os vértices em vermelho são articulações



Articulações e busca em profundidade

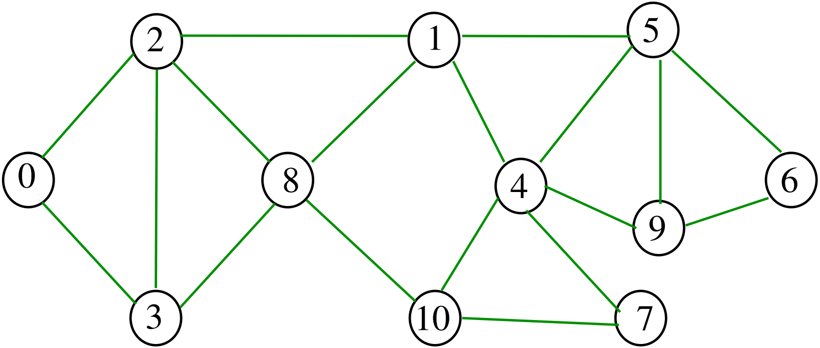
É possível encontrar todas as articulações de um grafo através de uma variante da função `bridgeR`

Exemplo: os vértices em vermelho são articulações



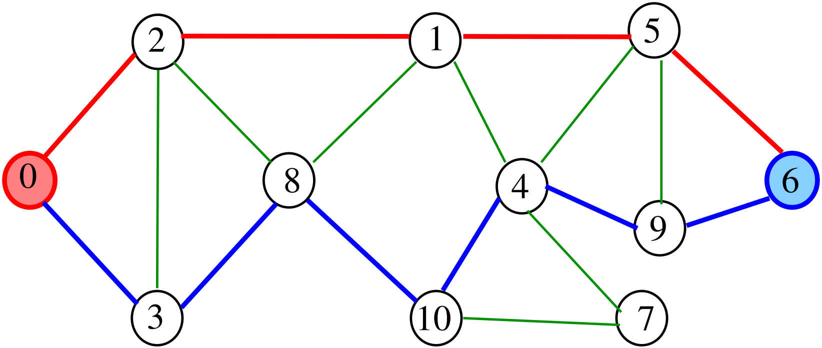
Biconexão

Um grafo é **biconexo** (= *biconnected*) ou **2-conexo** se é conexo e não tem articulações



Fato básico

Um grafo é biconexo se e somente se, para cada par (`s`,`t`) de vértices, existem (pelo menos) dois caminhos de `s` a `t` sem vértices internos em comum



Componentes fortemente conexos

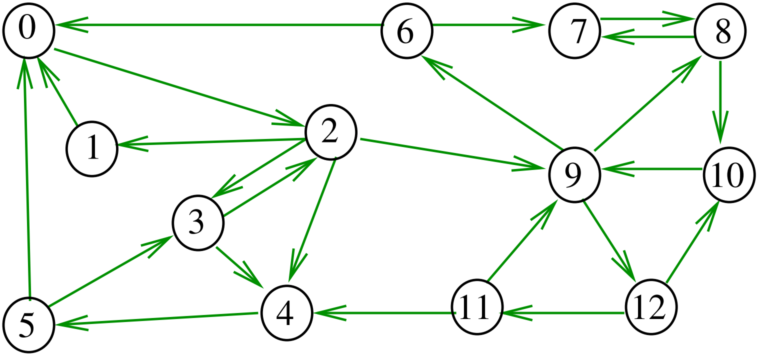
S 19.8

CLRS 22.5

Digrafos fortemente conexos

Um digrafo é **fortemente conexo** se e somente se para cada par {`s`,`t`} de seus vértices, existem caminhos de `s` a `t` e de `t` a `s`

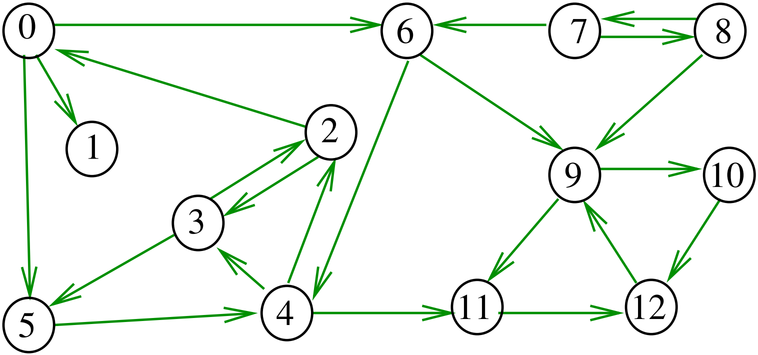
Exemplo: um digrafo fortemente conexo



Componentes fortemente conexos

Um componente **fortemente conexo** (= *strongly connected*) é um conjunto maximal de vértices `W` tal que digrafo induzido por `W` é fortemente conexo

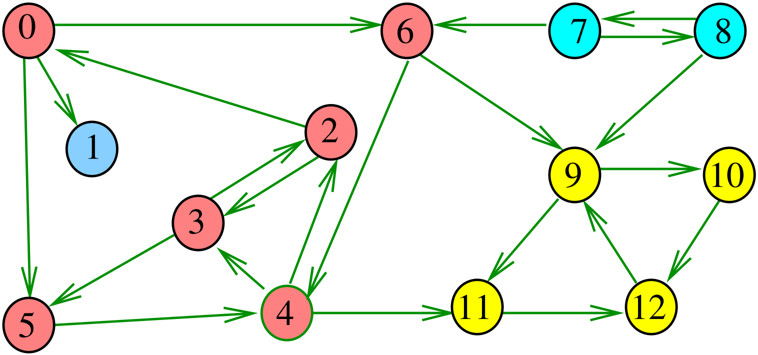
Exemplo: 4 componentes fortemente conexos



Componentes fortemente conexos

Um componente **fortemente conexo** (= *strongly connected*) é um conjunto maximal de vértices `W` tal que digrafo induzido por `W` é fortemente conexo

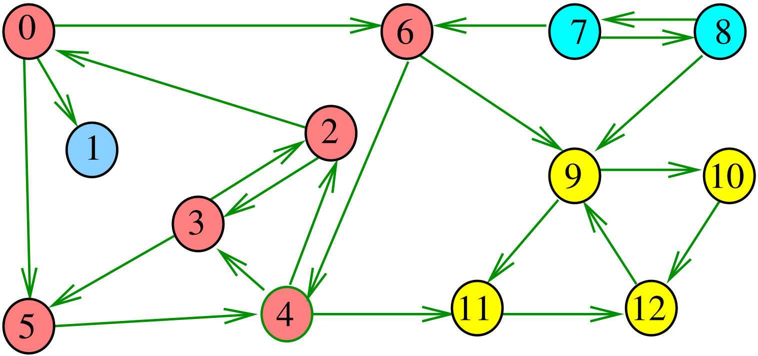
Exemplo: 4 componentes fortemente conexos



Determinando componentes f.c.

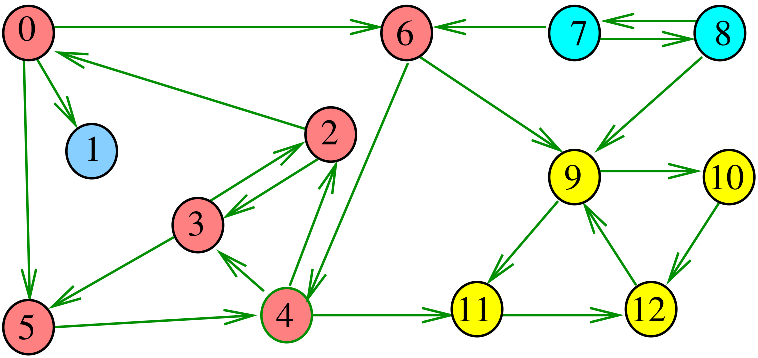
Problema: determinar os componentes fortemente conexos

Exemplo: 4 componentes fortemente conexos



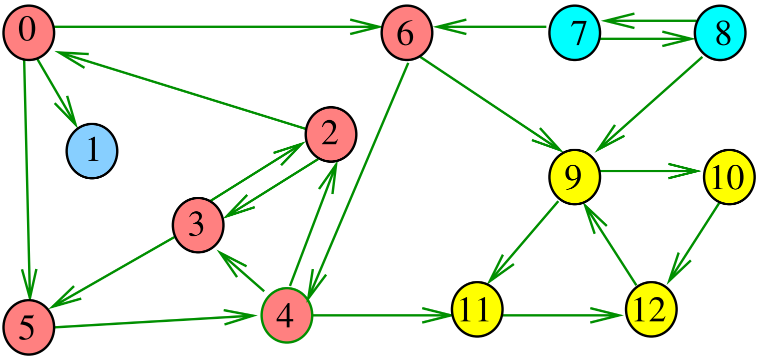
Exemplo

v	0	1	2	3	4	5	6	7	8	9	10	11	12
sc[v]	2	1	2	2	2	2	2	3	3	0	0	0	0



Exemplo

v	0	1	2	3	4	5	6	7	8	9	10	11	12
sc[v]	2	1	2	2	2	2	2	3	3	0	0	0	0



strongreach

```
int
strongreach(Digraph G,Vertex s,Vertex t)
{
   return sc[s]==sc[t];
}
```

Força Bruta

```
  int DIGRAPHsc1 (Digraph G) {
     Vertex v, w; int n;
     Graph H = GRAPHinit(G->V);
1    for (v = 0; v < G->V; v++)
2        for (v = w+1; v < G->V; v++)
3            if (DIGRAPHpath(G,v,w)==1
                 && DIGRAPHpath(G,w,v)==1)
4                GRAPHinsertE(H,v,w);
5    n = GRAPHcc(H);
6    for (v = 0; v < G->V; v++) sc[v]=cc[v];
7    return n;
  }
```

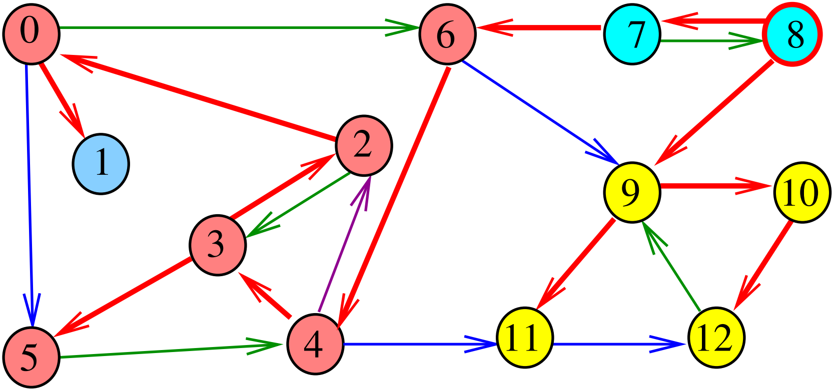
Consumo de tempo

O consumo de tempo da função `DIGRAPHsc1` para vetor de listas de adjacência é $O(V^2(V + A))$.

O consumo de tempo da função `DIGRAPHsc1` para matriz de adjacência é $O(V^4)$.

Propriedade

Vértices de de um componente fortemente conexo é uma **subarborescência** em uma floresta DFS



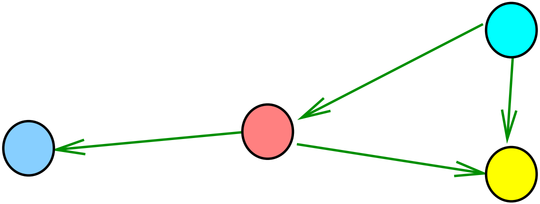
Digrafos dos componentes

O **digrafo dos componentes** de G tem um vértice para cada componente fortemente conexo e um arco U-W se G possui um arco com ponta inicial em U e ponta final em W

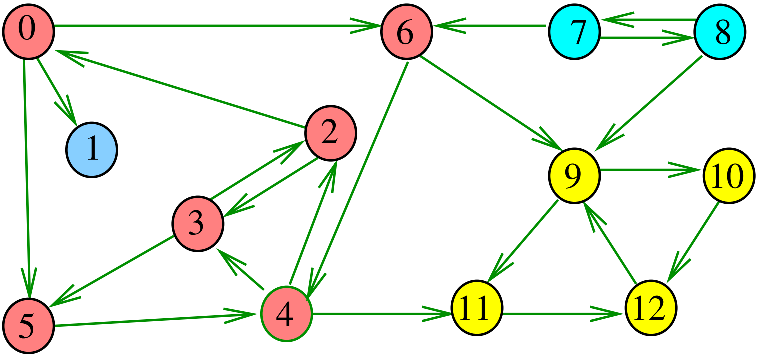
Digrafos dos componentes

O **digrafo dos componentes** de G tem um vértice para cada componente fortemente conexo e um arco U-W se G possui um arco com ponta inicial em U e ponta final em W

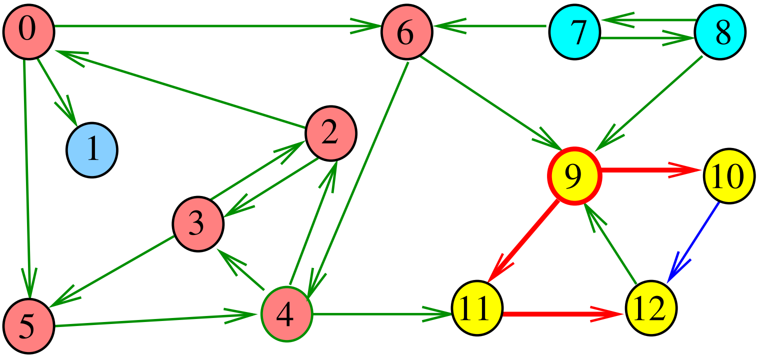
Digrafo dos componente é um DAG



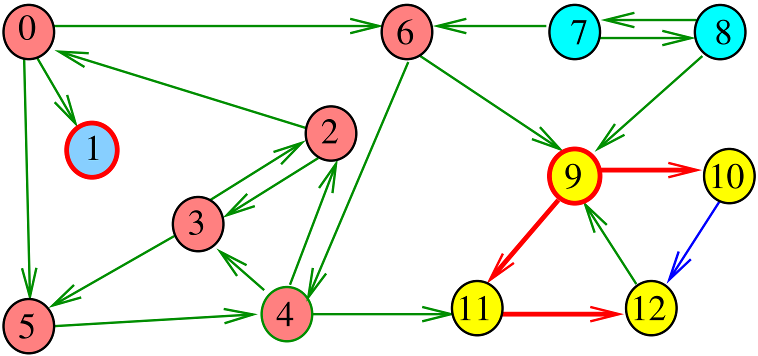
Idéia . . . G e DFS



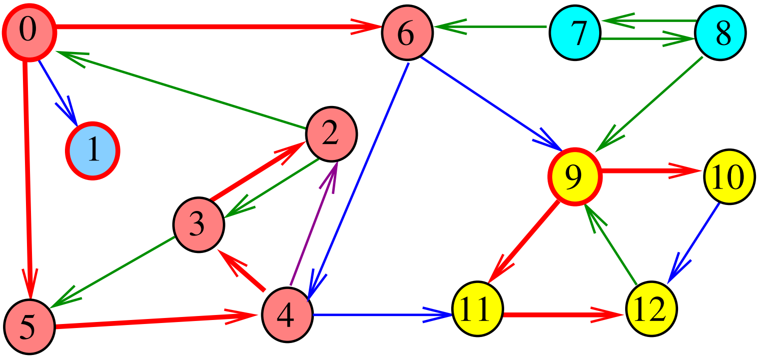
Idéia . . . G e DFS



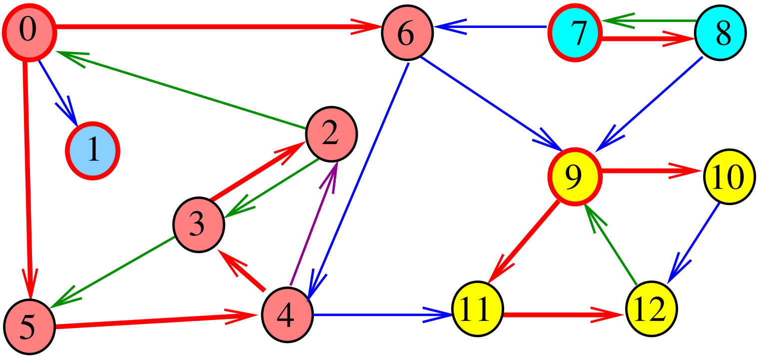
Idéia . . . G e DFS



Idéia ... G e DFS



Idéia ... G e DFS

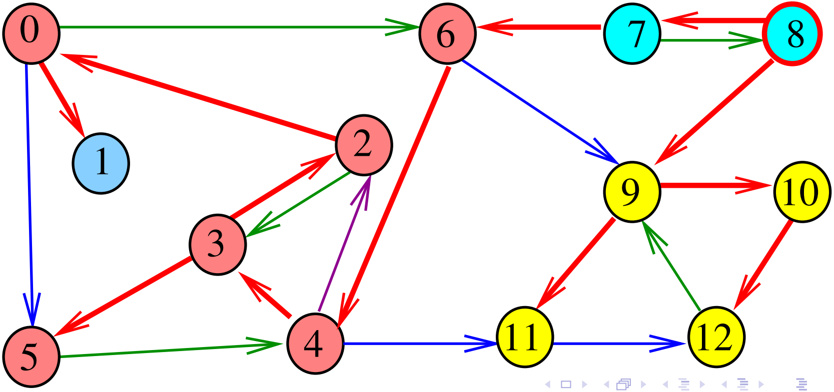


Numeração pós-ordem

pos[v] = numeração pós-ordem de v

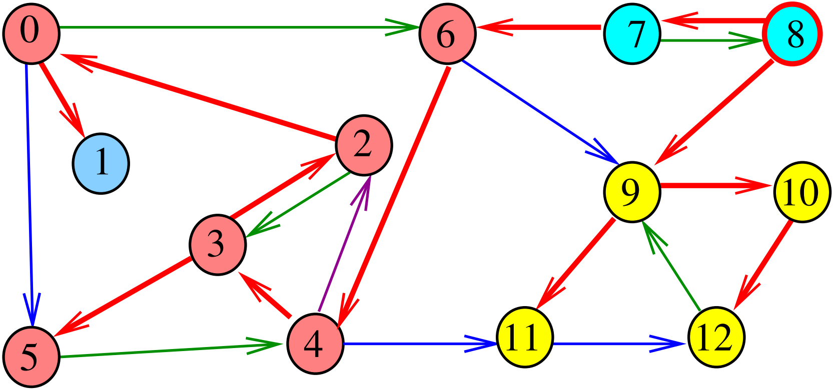
sop[i] = vértice de numeração pós-ordem i

pos[W] = maior numeração pós-ordem de um vértice em W



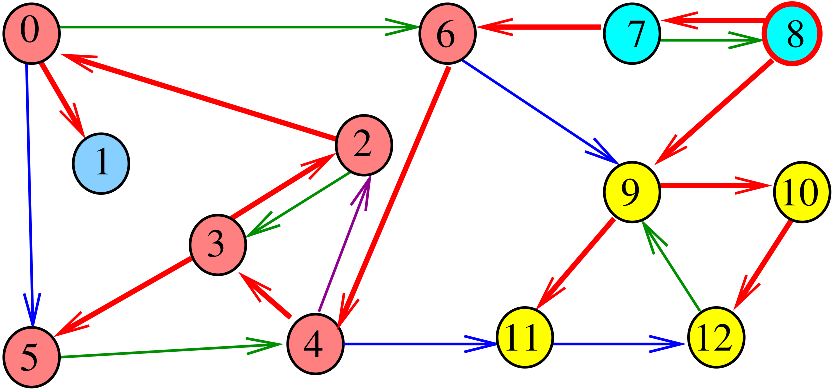
Exemplo

v	0	**1**	2	3	4	5	**6**	7	**8**	**9**	10	11	12
pos[v]	6	5	7	8	9	4	10	11	12	3	1	2	0



Exemplo

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sop[i]	12	10	11	**9**	5	**1**	0	2	3	4	**6**	7	**8**



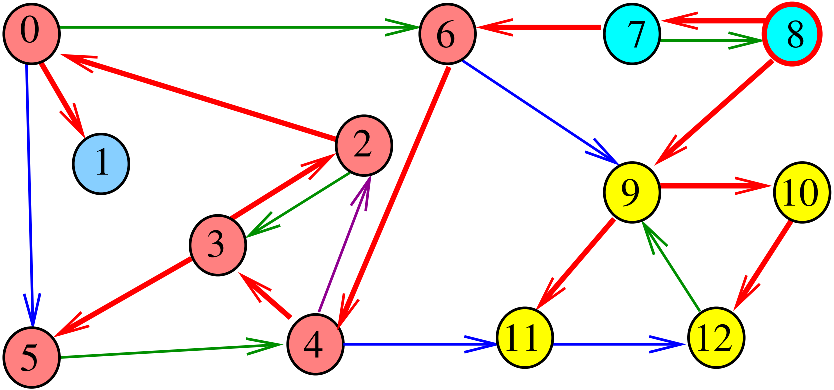
Exemplo

$\texttt{pos}[\{7, \mathbf{8}\}] = 12$

$\texttt{pos}[\{0, 2, 3, 4, 5, \mathbf{6}\}] = 10$

$\texttt{pos}[\{\mathbf{1}\}] = 5$

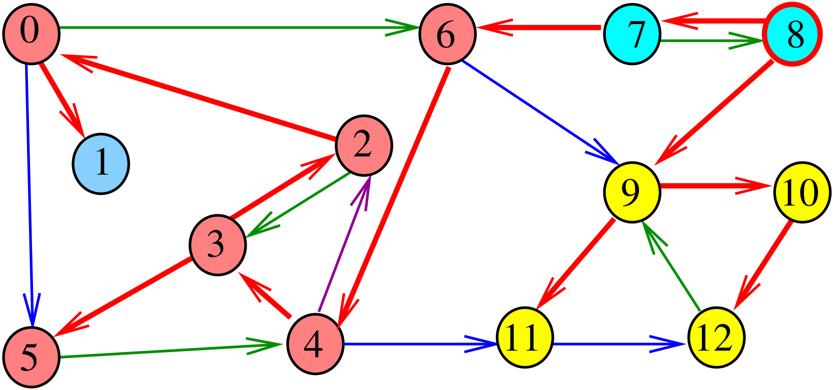
$\texttt{pos}[\{\mathbf{9}, 10, 11, 12\}] = 3$



Numeração pós-ordem e componentes f.c.

Se `U` e `W` são componentes f.c. e existe arco com ponta inicial em `U` e ponta final em `W`, então

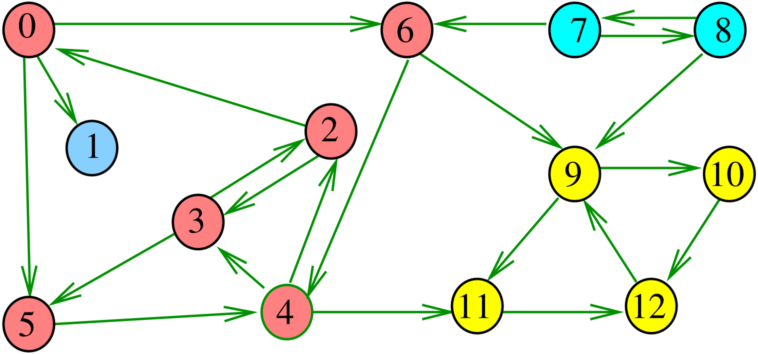
$$\texttt{pos}[\texttt{U}] > \texttt{pos}[\texttt{W}]$$



Propriedade

Um digrafo G e seu digrafo reverso R têm os mesmos componente fortemente conexos

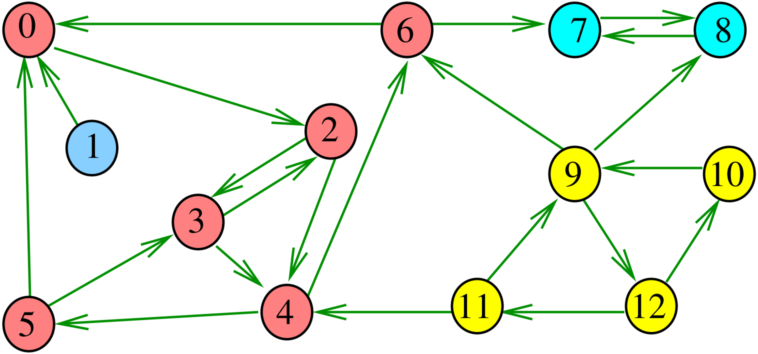
Exemplo: Digrafo G



Propriedade

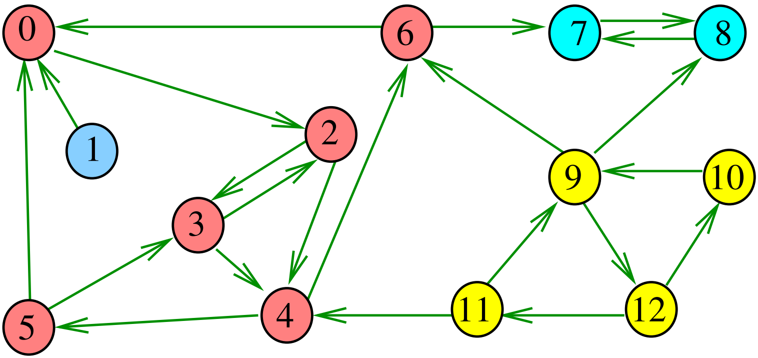
Um digrafo G e seu digrafo reverso R têm os mesmos componente fortemente conexos

Exemplo: Digrafo reverso R de G



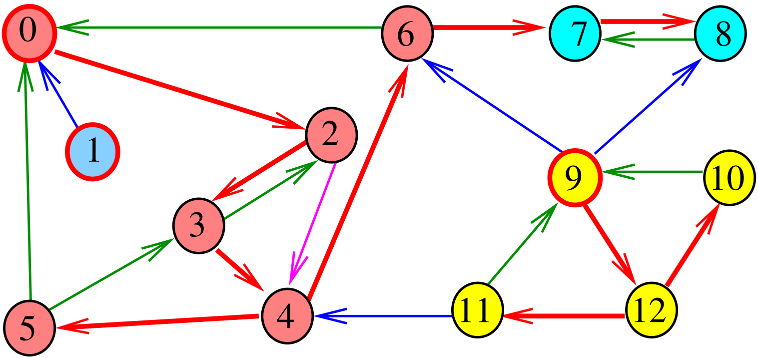
Digrafo reverso R

v	0	1	2	3	4	5	6	7	8	9	10	11	12
sc[v]	2	1	2	2	2	2	2	3	3	0	0	0	0



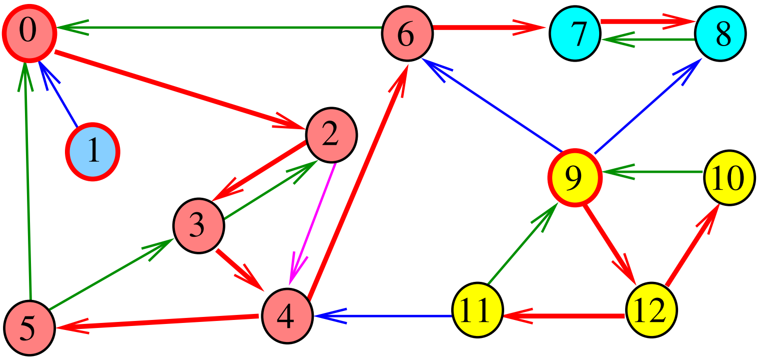
Digrafo reverso R e DFS

v	**0**	**1**	2	3	4	5	6	**7**	8	**9**	10	11	12
pos[v]	7	8	6	5	4	3	2	1	0	12	9	10	11



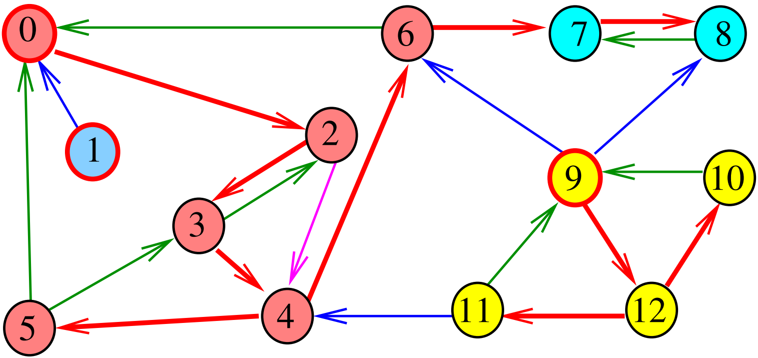
Digrafo reverso R e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sop[i]	8	**7**	6	5	4	3	2	**0**	**1**	10	11	12	**9**



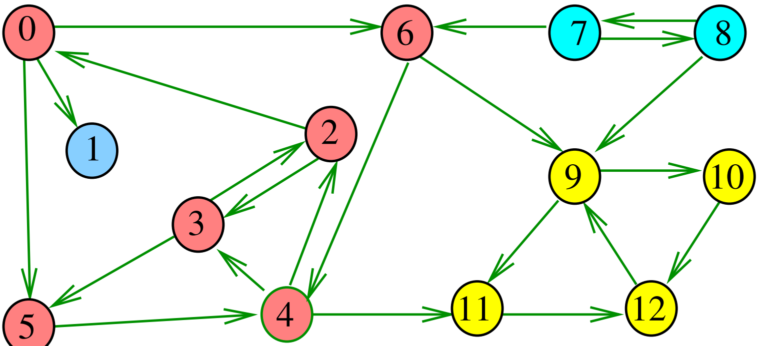
Digrafo reverso R e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sop[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



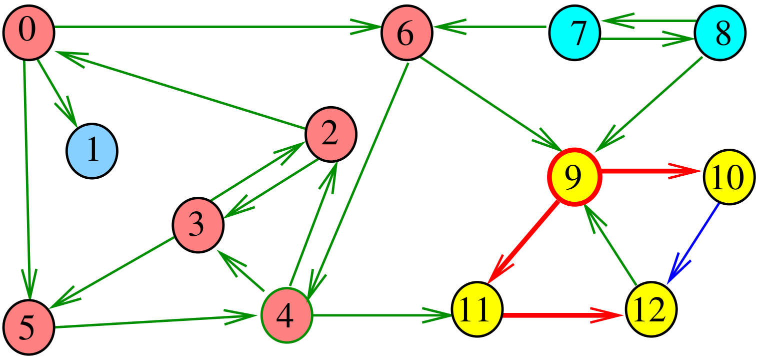
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sop[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



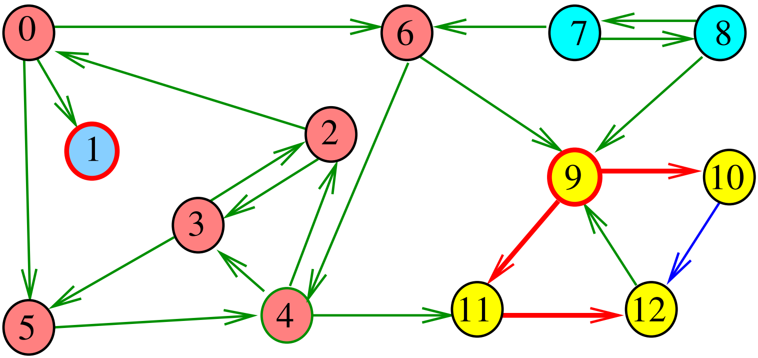
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sop[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



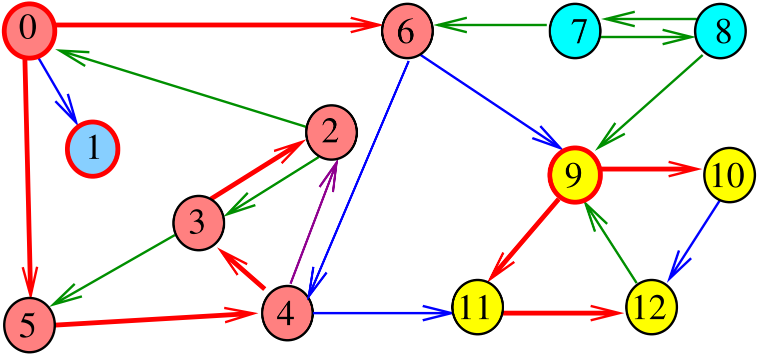
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sop[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



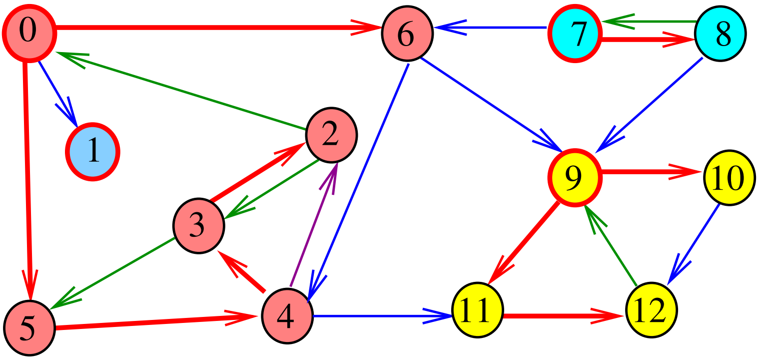
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sop[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sop[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



Algoritmo de Kosaraju

A função devolve o número de componentes fortemente conexos do digrafo `G`

```
static int sc[maxV];
static Vertex sop[maxV], sopR[maxV];
static int cnt, id;
```

Além disso, ela armazena no vetor `sc` o número do componente a que o vértice pertence: se o vértice `v` pertence ao `k`-ésimo componente então `sc[v] == k-1`

```
int DIGRAPHsc (Graph G)
```

DIGRAPHsc

```
int DIGRAPHsc (Digraph G) {
 Vertex v;
 int id, i;
 Digraph R = DIGRAPHreverse(G);
 cnt = 0;
 for (v = 0; v < R->V; v++) sc[v] = -1;
 for (v = 0; v < R->V; v++)
     if (sc[v] == -1)
         dfsRsc(R, v, 0);
```

DIGRAPHsc

```
for (i = 0; i < G->V; i++)
    sopR[i] = sop[i];
cnt = id = 0;
for (v = 0; v < G->V; v++) sc[v] = -1;
for (i = G->V-1; i > 0; i--)
    if (sc[sopR[i]] == -1)
        dfsRsc(G, sopR[i], id++);
DIGRAPHdestroy(R);
return id;
}
```

dfsRsc

```
void dfsRsc(Digraph G,Vertex v,int id){
 link p;
 sc[v] = id;
 for (p=G->adj[v];p!=NULL;p=p->next)
     if (sc[p->w] == -1)
         dfsRsc(G, p->w, id);
 pos[v] = cnt; /* não precisa */
 sop[cnt++] = v;
}
```

DIGRAPHreverse

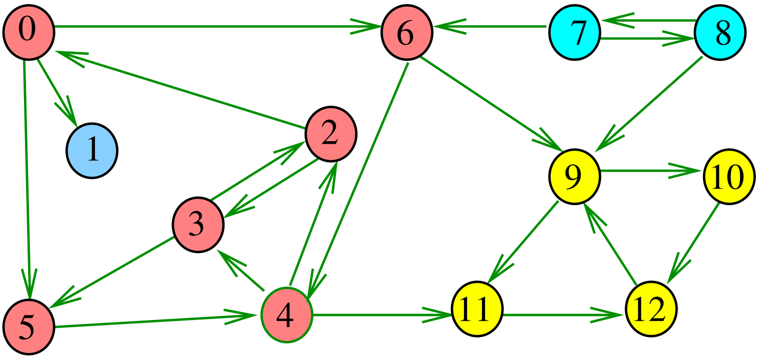
```
Digraph DIGRAPHreverse (Digraph G) {
 Vertex v; link p;
 Digraph R= DIGRAPHinit(G->V);
 for (v=0; v < G->V; v++)
     for (p=G->adj[v];p!=NULL; p=p->next)
         DIGRAPHinsertA(G,p->w,v);
 return R;
}
```

Consumo de tempo

O consumo de tempo da função DIGRAPHsc é $O(V + A)$.

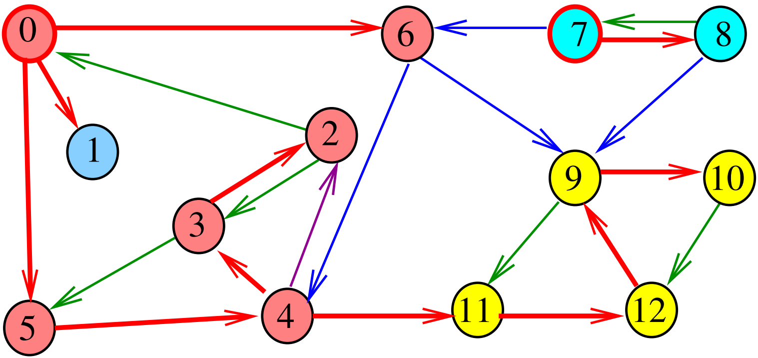
Algoritmo de Tarjan

O menor **número de pré-ordem** de um vértice "ativo" que pode ser alcançado por `v` utilizando arcos da arborescência e **até um** arco de retorno será denotado por `low[v]`



Exemplo

v	**0**	**1**	2	3	4	5	6	**7**	8	9	10	**11**	12
pre[v]	0	9	4	3	2	1	10	11	12	7	8	5	6
low[v]	0	9	0	0	0	0	0	11	11	5	6	5	5



DIGRAPHsc

```
void DIGRAPHsc (Graph G) {
Vertex v;
cnt = id = t = 0;
for (v = 0; v < G->V; v++)
    pre[v] = -1;
for (v = 0; v < G->V; v++)
    if (pre[v] == -1)
        dfsRsc(G, v);
}
```

```
void dfsRsc(Digraph G,Vertex v){
    link p; Vertex w; int min;
    pre[v] = cnt++; low[v] = pre[v];
    min = low[v]; s[t++] = v;
    for (p=G->adj[v];p!=NULL;p=p->next){
        if (pre[w=p->w]==-1) dfsRsc(G,w);
        if (low[w] < min) min=low[w];
    }
    if (min<low[v]) {low[v]=min; return;}
    do {
        sc[w=s[--t]]=id; low[w]=G->V;
    } while (s[t] != v);
    id++;
}
```

```
void dfsRsc(Digraph G,Vertex v){
   link p; Vertex w;
   pre[v] = cnt++; low[v] = pre[v];
   s[t++] = v;
   for (p=G->adj[v];p!=NULL;p=p->next){
        if (pre[w=p->w]==-1) dfsRsc(G,w);
        if (low[w] < low[v]) low[v]=low[w];
   }
   if (low[v]<pre[v]) return;
   do {
        sc[w=s[--t]]=id; low[w]=G->V;
   } while (s[t] != v);
   id++;
}
```