## p.280 8.7 A formatted version

We have a network of computers and a list of bi-directional connections. Each of these connections allows a file transfer from one computer to another. Is it possible to send a file from any computer on the network to any other?

### Input:

Input consists of several test cases.
For each test case, the first line contains N (<=10,000), the total number of computers in a network. Each computer in the network is then represented by a positive integer between 1 and N.
Then in the following lines, the input is given in the format:

**I c1 c2**

where I stands for inputting a connection between c1 and c2; or

**C c1 c2**

where C stands for checking if it is possible to transfer files between c1 and c2; or

**S**

where S stands for stopping this case.

### Output:

For each C case, print in one line the word "yes" or "no" if it is possible or impossible to transfer files between c1 and c2, respectively.

At the end of each case, print in one line "The network is connected." if there is a path between any pair of computers; or "There are k components." where k is the number of connected components in this network.

Print a blank line between test cases.

| Sample Input: | Sample Output: |
|---|---|
| 3<br>C 1 2<br>I 1 2<br>C 1 2<br>S<br>3<br>I 3 1<br>I 2 3<br>C 1 2<br>S | no<br>yes<br>There are 2 components.<br><br>yes<br>The network is connected. |

Note: Must use union-by-size and path compression.

## Sample code:

```
/* an on-line algorithm for checking file transfer ability */
/* key-functions: SetUnion and Find */

#include <stdio.h>
#include <string.h> /* for checking input only */

#define NumSets 10000

typedef int DisjSet[ NumSets + 1 ];
typedef int SetType;
typedef int ElementType;

void Initialization( DisjSet S , int n)
{ /* initialize disjoint sets for union-by-size*/
    int   i;

    for ( i = n; i > 0; i -- )
        S[i] = -1;
}

void SetUnion( DisjSet S, SetType Root1, SetType Root2 )
{   /* Union-by-size */
    /* Root1 and Root2 must be roots */
    if ( S[Root2] < S[Root1] ) { /* if Root2 is larger */
        S[Root2] += S[Root1];  /* add Root1 to Root2 */
        S[Root1] = Root2;      /* make Root2 the new root */
    }
    else {                     /* if Root1 is not smaller */
        S[Root1] += S[Root2];  /* add Root2 to Root1 */
        S[Root2] = Root1;      /* make Root1 the new root */
    }
}

SetType Find( ElementType X, DisjSet S )
{ /* Find with path compression */
    if ( S[X] <= 0 )  /* root found */
        return X;         /* return the root */
    else
        return S[X] = Find( S[X], S ); /* compress and return root */
}
```

```
void Input_connection( DisjSet S )
{ /* read a connection and reset the sets if necessary */
     ElementType u, v;
     SetType Root1, Root2;

     scanf("%d %d\n", &u, &v);

     Root1 = Find(u, S);
     Root2 = Find(v, S);
     if ( Root1 != Root2 )    /* if u and v are not connected */
          SetUnion( S, Root1, Root2 );    /* then connect them */
}

void Check_connection( DisjSet S )
{ /* check if file transfer between two computers is possible */
     ElementType u, v;
     SetType Root1, Root2;

     scanf("%d %d\n", &u, &v);

     Root1 = Find(u, S);
     Root2 = Find(v, S);
     if ( Root1 == Root2 )    /* if u and v are connected */
          printf("yes\n");
     else
          printf("no\n");
}

void Check_network( DisjSet S , int n )
{
     int    i, counter = 0;

     for (i=1; i<=n; i++) {
          if ( S[i] < 0 )
               counter ++;    /* count one root */
     }
     if ( counter == 1 )
          printf("The network is connected.\n");
     else
          printf("There are %d components.\n", counter);
}
```

```
int main( )
{
    DisjSet S;
    int n, flag=0;
    char in;

    while (scanf("%d\n", &n) != EOF) {    /* for each test case */
        if (flag)    /* if it's not the 1st case */
            printf("\n");    /* print a blank line before output */
        else    /* if it is the 1st case */
            flag = 1;    /* output without the blank line, and mark the flag */
        Initialization( S, n );
        do {    /* for each command */
            scanf("%c", &in);
            switch (in) {
                case 'I': Input_connection( S ); break;
                case 'C': Check_connection( S ); break;
                case 'S': Check_network( S, n ); break;
            }
        } while ( in != 'S');
    }

    return 1;
}
```

## p.337 9.2

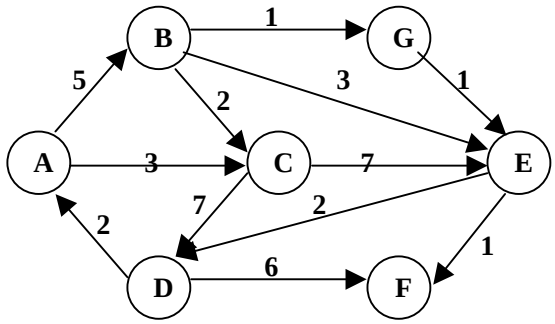
**If a stack is used instead of a queue for the topological sort algorithm in Section 9.1, does a different ordering result? Why might one data structure give a "better" answer?**

## Answer:

Using a stack is the depth first in topologically sorting, while using a queue is the breadth first sorting. So they have different ordering result.
Using a stack or a queue can avoid the sequential scan through all the vertices in the search for a vertex of indegree 0.
A stack, for example, schedules courses by branches, while a queue by levels.

## p.337 9.5

a. Find the shortest path from A to all other vertices for the graph in Figure 9.80.
b. Find the shortest unweighted path from B to all other vertices for the graph in Figure 9.80.




**a.**

| source | destination | path | cost |
|---|---|---|---|
| A | B | AB | 5 |
| | C | AC | 3 |
| | D | ABGED | 9 |
| | E | ABGE | 7 |
| | F | ABGEF | 8 |
| | G | ABG | 6 |

**b.**

| Source | Destination | path | cost |
|---|---|---|---|
| B | A | BEDA | 3 |
| | C | BC | 1 |
| | D | BCD | 2 |
| | E | BE | 1 |
| | F | BEF | 2 |
| | G | BG | 1 |

## p.338 9.10

a. Explain how to modify Dijkstra's algorithm to produce a count of the number of different minimum paths from v to w.
b. Explain how to modify Dijkstra's algorithm so that if there is more than one minimum path from v to w, a path with the fewest number of edges is chosen.

## Answer a:

```
void Dijkstra( Table T )
{ /* T[ ].Count is initialized to be 0 */
        vertex v, w;
        for ( ; ; ) {
            v = smallest unknown distance vertex;
            if ( v == NotAVertex )
                break;
            T[v].Known = True;
            for ( each w adjacent to v )
                if( !T[w].Known )
                    if( T[v].Dist + Cvw < T[w].Dist ) {
                        Decrease( T[w].Dist to T[v]+Cvw )
                        T[w].Path = v;
                        T[w].Count = 1;
                    }
                    else if( T[v].Dist + Cvw == T[w].Dist )
                            T[w].Count += 1;
        }
}
```

## Answer b:

```
void Dijkstra( Table T )
{ /* T[ ].Count is initialized to be 0 */
        vertex v, w;
        for ( ; ; ) {
            v = smallest unknown distance vertex;
            if ( v == NotAVertex )
                break;
            T[v].Known = True;
            for ( each w adjacent to v )
                if( !T[w].Known )
                    if( T[v].Dist + Cvw < T[w].Dist ) {
                        Decrease( T[w].Dist to T[v]+Cvw )
                        T[w].Path = v;
                        T[w].Count = T[v].Count + 1;
                    }
                    else if( ( T[v].Dist + Cvw == T[w].Dist )
                              && ( T[v].Count + 1 < T[w].Count ) )
                            T[w].Count = T[v].Count + 1;
        }
}
```