Consider the following pseudo code of a function which takes an integer $n \geq 0$ as input.

```
Function $foo(n)$
| if $n$==0 then
| | Return;
| end
| for $i = 0$ to $n$ - 1 do
| | Print '*';
| end
| $foo(n-1)$;
```

Let $T(n)$ be the number of times the above function prints a star (*) when called with input $n \geq 0$. What is $T(n)$ exactly, in terms of only n (and not values like $T(n1)$ or $T(n2)$)? Prove your statement.

$T(0) = 0$ star
$T(1) = 1$ star
$T(2) = 3$ star
$T(3) = 6$ star
$T(n) = \sum_{1=0}^{n} i = \frac{n(n+1)}{2}$ star

Proof by Induction

$T(n) =$ The number of stars generated by the function can be modeled by the equation $\sum_{1=0}^{n} i$

Base Case: When n = 0, then:
Following the function will return directly with no printed stars.
Following the equation, $T(0) = 0$ stars
Therefore, the base case is verified

Induction Step: If $T(k)$ is true then prove $T(k+1)$:
According to the function, $T(k+1)$ will call stars $k+1$ times followed by a function call to $T((k+1)-1) = T(k) = \sum_{1=0}^{k} i$.
Therefore:
$T(k+1) = (k+1) + \sum_{1=0}^{k} i$
$T(k+1) = \sum_{1=0}^{(k+1)} i$

Thus the induction step is satisfied and by the principle of mathematical induction, the equation holds true for the function.

Same as Problem 2 above, but for the following:

```
Function bar(n)
| Print '*';
| if n==0 then
| | Return;
| end
| for i = 0 to n - 1 do
| | bar(i);
| end
```

$T(0) = 1$ star
$T(1) = 2$ star
$T(2) = 4$ star
$T(3) = 8$ star
$T(n) = 2^n$ star

Proof by Induction

$T(n) =$ The number of stars generated by the function can be modeled by the equation 2^n

Base Case: When n = 0, then:
Following the function will print one star and then end by a return.
Following the equation, $T(0) = 1$ star
Therefore, the base case is verified

Induction Step: If $T(k)$ is true then prove $T(k+1)$:
$T(k+1) = 1 + 2^k + 2^k - 1$
This is made of three components which add up to the sum of stars.
The first is a 1 because every call to *bar* will generate one star.
The second is 2^k because there will be 2^k additional calls to *bar*.
The third is $2^k - 1$ because 2^k *bar* calls will cause $2^k - 1$ additional calls.

This equation will then be simplified $T(k+1) = 2^k + 2^k$
$T(k+1) = 2 * 2^k$
$T(k+1) = 2^{(k+1)}$

Thus the induction step is satisfied and by the principle of mathematical induction, the equation holds true for the function.

You are given two connected undirected graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, with $V_1 \cap V_2 = \emptyset$, as well as a node $s \in V_1$ and a node $t \in V_2$. You are also given a set of edges E' which you are able to build: each edge in E' has one endpoint in V_1 and one endpoint in V_2. Thus, building any edge of E' would connect the two graphs together, and in particular would form paths connecting s and t. You are asked to determine an edge $e \in E'$ whose addition to the graphs G_1 and G_2 would result in the shortest possible (i.e., smallest number of edges) distance between s and t among all the edges in E'. Give a linear-time algorithm to solve this problem.

Just give the algorithm, no proofs are necessary. Recall that linear-time means linear in the size of the input, which for this case is the number of nodes plus the number of edges in the graphs G_1 and G_2, plus the number of edges in E'. Hint: you can use BFS as a subroutine to solve this.

The algorithm would consist of three distinct parts.
The first is to take a BFS from s to find the distance to all nodes in V_1.
The first is to take a BFS from t to find the distance to all nodes in V_2.
There is also a need for two placeholder variables to store distance and edge which we will call *dist* and *ed* The third is to loop for all e in E'. Inside the loop we will look through each e to find the distance from s to t by adding up the distance from s in one node connected to e in G_1 and the distance from t in one node connected to e in G_2. This sum would be compared to the values stored in the placeholder *dist* and if it is shorter, would stroe the distance in *dist* and e in *ed*. This will continue through all the possible edges in E'.
The function can finally return the edge stored i *ed*.

DPV Problem 3.7(a). Give the algorithm and argue why it is correct. You can assume the graph is connected. Hint: you can use BFS to solve this.

A bipartite graph is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V_1 \cap V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set (for instance, if $u, v \in V_1$, then there is no edge between u and v).
Give a linear-time algorithm to determine whether an undirected graph is bipartite.

The algorithm to find whether the graph is bipartite using BFS consists of three components.

```
1 Start at some node $s$ and color it black (will use black and white as colors to denomonate $V_1$ and $V_2$). Following this put it in the BFS queue.
2 Loop while the BFS queue is not empty.
| Remove next value from the BFS queue and put it in variable $x$.
| loop through every neighboring node for $x$.
| | If the neighbor is not colored, color it the opposite color of $x$ and add it to the BFS queue.
| | If the neighbor is colored, and it is the opposite color of $x$ then continue.
| | If the neighbor is colored, and it is the same color as $x$ then return not bipartite.
3 Return it is bipartite.
```

Proving the correctness of the algorithm is done by showing that the algorithm does indeed return bipartite when it is and not bipartite when it is not.

bipartite $\rightarrow$ algorithm shows bipartite
Normally, the problem claims that there should be no edges between nodes in V_1 to V_1 and V_2 to V_2. Therefore, if we thought about it as colors, in the graph, the starting node s will be a color while its neighbors will be the opposite color and the neighbors of those will again be the first color, so on so forth. IF this is the case, according to the algorithm, it follows a very similar implementation in which it checks it will not return not bipartite because there wouldn't be a case with the same colors as neighbors.

not bipartite $\rightarrow$ algorithm shows not bipartite
If the graph is not bipartite, according to the problem, this represents two nodes within either V_1 or V_2 being connected. In terms of colors this would signify two neighbors of the same color. The algorithm will ensure this conditional due to the check for neighbors of the same color which will return not bipartite.

DPV Problem 3.11. Your algorithm should also return the length (number of edges) of the shortest cycle containing e, if one exists. Just give the algorithm, no proofs are necessary. Hint: you can use BFS to solve this.

Design a linear-time algorithm which, given an undirected graph G and a particular edge e in it, determines whether G has a cycle containing e.

A linear time algorithm to show whether or not e is part of a cycle or not is through BFS. e as an edge will have two nodes connected to it which will be called s and t. We can find whether e is part of a cycle or not through an algorithm that has two differences from a normal BFS.
The first is that we will remove e. Following this, the second is to conduct a BFS from s and check the nodes every time a node is to be added to the BFS queue.
If the node to be added is t, then the algorithm will return there is a cycle. Otherwise, if t is never found, it will return that there is o cycle. The length of the shortest cycle can also be found by returning the distance from s to t and adding one to represent edge e.