PROBLEM OF THE MONTH

POM

2018 - 2019

December, 2018

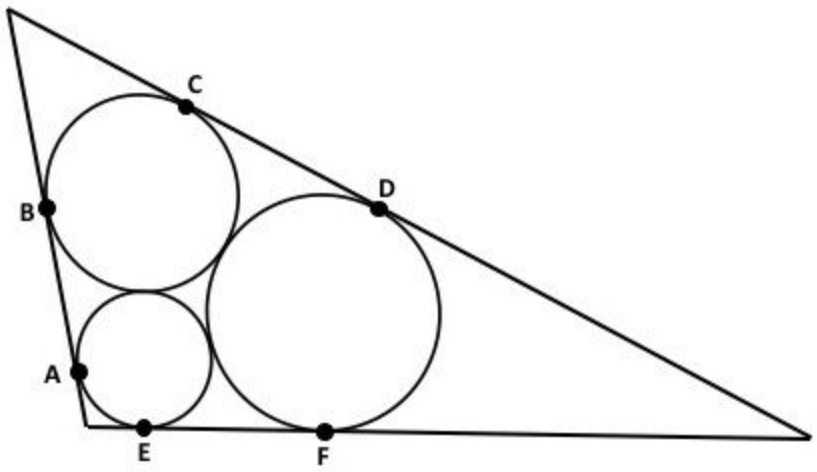
MATHEMATICS

5 points:

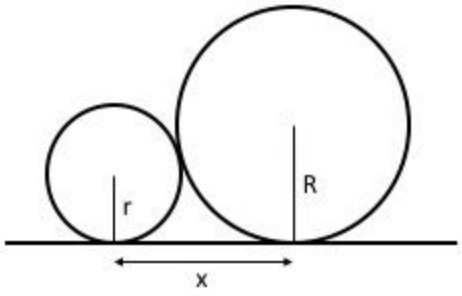
Three circles of radii R_1, R_2, and R_3 are pairwise externally tangent to each other, as in the diagram.
It is known that one can form a right triangle from the segments AB, CD, and EF.

Find R_1 given that $R_2 = 3$ and $R_3 = 4$.

(Note that there may be more than one correct answer.)



Hint:



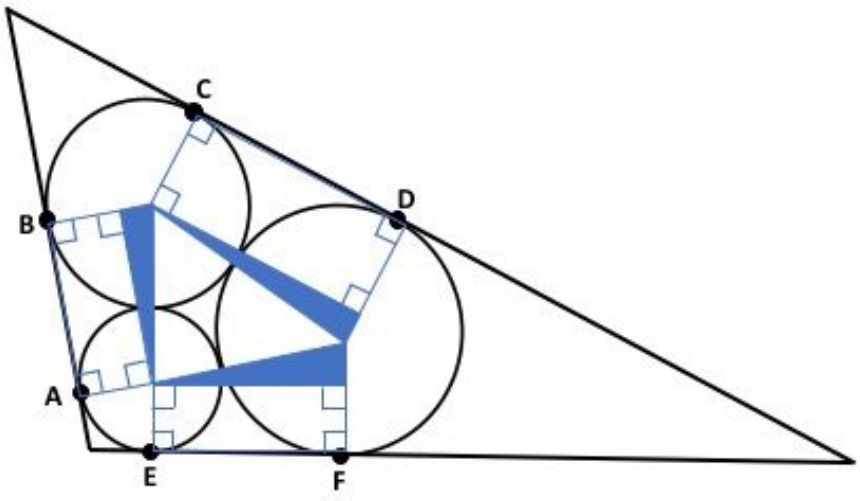
Find x in terms of R and r, and then apply it to the problem to write the pythagorean relation between AB, CD, and EF.

Answer:

$R_1 = 12/7$ or $R_1 = 12$.

Solution:

In the image below, there are three blue right triangles and three rectangles. The hypotenuse of each triangle connects the centers of two circles, so its length is the sum of their radii. The shorter leg of each triangle is the difference between the radii.



Using this knowledge and the pythagorean theorem, it is possible to find the length of the longer leg of each triangle, which are equal to the segments AB, CD, and EF.

We have

$(R_1 + R_2)^2 - (R_1 - R_2)^2 = 4R_1R_2 = X^2$
so that $X = 2\sqrt{R_1R_2}$.
Similarly, the lengths of the other two segments are $2\sqrt{R_2R_3}$ and $2\sqrt{R_3R_1}$.

These three segments form a right triangle, so the relationship between them is given by the pythagorean theorem, except we don't know which segment is the hypothesis.

Plugging in the numbers, we have $2\sqrt{3R_1}$, $2\sqrt{4R_1}$, and $2\sqrt{12}$. We can ignore the factor of 2 because it will cancel in the equation.

$\sqrt{3R_1}$ is definitely shorter than $\sqrt{4R_1}$, so it can't be the hypotenuse. $\sqrt{4R_1}$ and $\sqrt{12}$, however, could both work as a hypotenuse because we don't know which is larger.

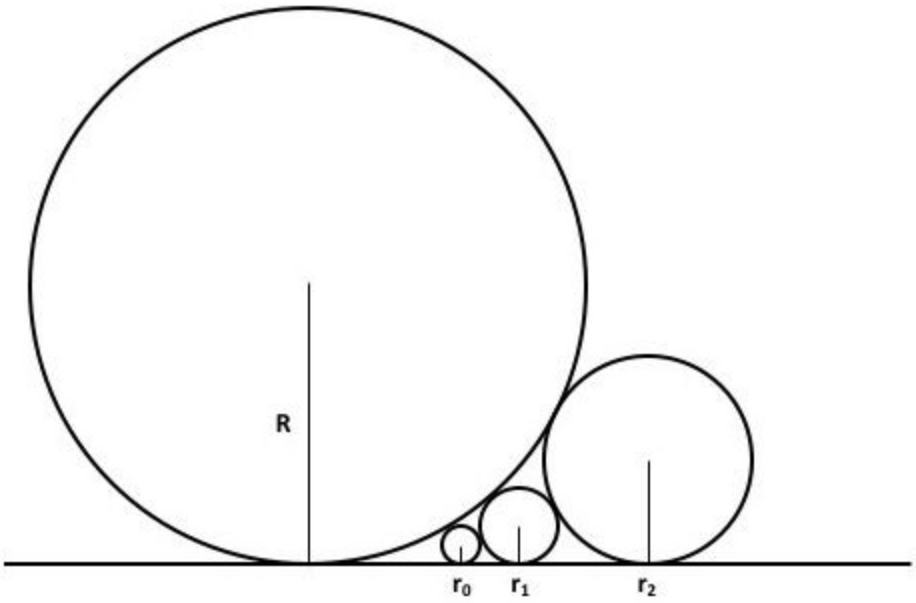
So we have:
$3R_1 + 4R_1 = 12$
or
$3R_1 + 12 = 4R_1$

Both are valid, and the solutions are $R_1 = 12/7$ or $R_1 = 12$.

10 points:

A large circle with radius R is tangent to a line. Smaller circles with radii r_0, r_1, r_2, etc. are constructed such that each is externally tangent to the large circle, the line, and its neighboring smaller circles, as in the figure.
It is known that $r_0 = R / 2018$.

How many such smaller circles can be constructed?
What is the radius of the last one of these circles?



Hint:

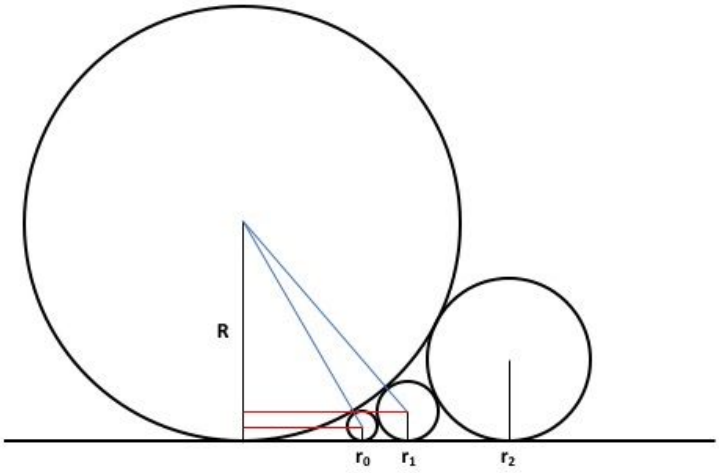
Find the relation between r_k, r_{k+1}, and R.

Answer:

45 to the first question

$R \times (\frac{1}{\sqrt{2018}-44})^2 \approx 1.18R$ to the second

Solution:



In the image above, the two triangles each made up of a red line, a blue line, and a segment of the radius R are right triangles.
The lengths of red segments, from the pythagorean theorem (see the solution of the previous problem), are $\sqrt{4Rr_0}$ and $\sqrt{4Rr_1}$

Their difference is $\sqrt{4Rr_1} - \sqrt{4Rr_0} = \sqrt{4r_0r_1}$

Rearranging, we get $\frac{1}{\sqrt{r_1}} = \frac{1}{\sqrt{r_0}} - \frac{1}{\sqrt{R}}$

This repeats for the next circle, etc so we have
$\frac{1}{\sqrt{r_n}} = \frac{1}{\sqrt{r_{n-1}}} - \frac{1}{\sqrt{R}}$
Iterating we obtain
$\frac{1}{\sqrt{r_n}} = \frac{1}{\sqrt{r_{n-1}}} - \frac{1}{\sqrt{R}} = \frac{1}{\sqrt{r_{n-2}}} - \frac{2}{\sqrt{R}} = \ldots = \frac{1}{\sqrt{r_0}} - \frac{n}{\sqrt{R}}$

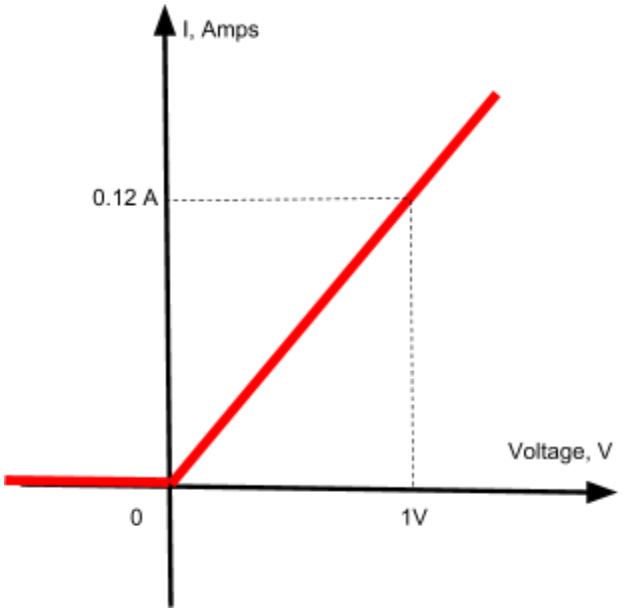
Now, using the fact that $r_0 = R/2018$, we can plug that in and we get
$\frac{1}{\sqrt{r_n}} = \frac{\sqrt{2018} - n}{\sqrt{R}}$
and
$\sqrt{r_n} = \frac{\sqrt{R}}{\sqrt{2018} - n}$

From this equation, it is clear that n cannot be larger than $\sqrt{2018}$. The largest n is therefore 44 and the maximal number of circles is 45 (numbers start from 0).
The largest circle is for n = 44, $r_n = R \times \left(\frac{1}{\sqrt{2018} - 44}\right)^2 \approx 1.18R$.

PHYSICS

5 points:

A Christmas tree is lighted by 55 light-emitting diodes (LEDs) connected in series and plugged directly into a to regular, 110V electric outlet. The outlet supplies an alternating current (AC): half of the time it flows in one direction and half of the time - in the opposite. The direction of the current changes back and forth 50 times per second. LEDs, however, conduct the current only in one direction, and shut down when the opposite voltage is applied. The figure shows schematically how the current depends on voltage for each LED. What is the overall power consumed by all the LEDs?



Hint: You can find AC voltage on each LED, and the current that corresponds to it. Note LEDs are off 50% of the time.

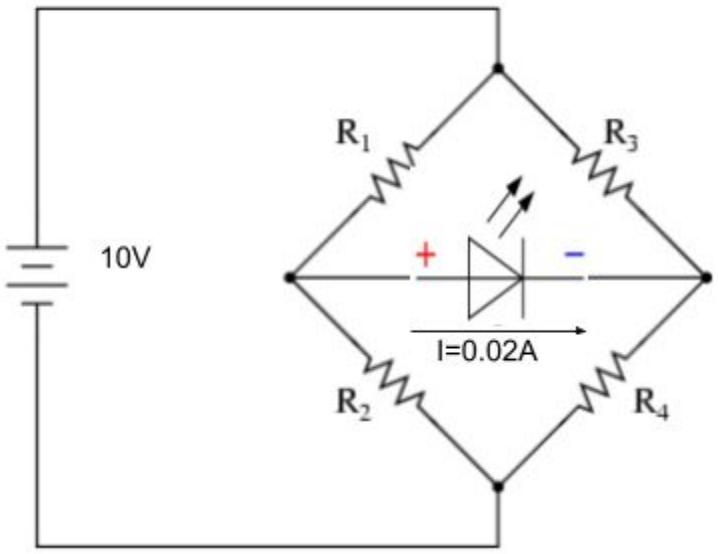
Answer: 13.2 Watt

Solution:

To find current we find AC voltage on each LED , **V=110V/55=2V.** This corresponds to **I=0.24A** (when LED is "ON"). For a regular light bulb current ***I*** is proportional to voltage ***V***. In that case, the power can be found as ***P=IV***. Unlike a light bulb, LEDs are "off" 50%of the time. Therefore, power is 50% less: $P = IV/2 = 13.2\ W$ **.**

10 points:

A light-emitting diode (LED) is plugged into the circuit shown in the figure. The ideal battery has voltage 10V, and three of the resistors all have the same resistances: $R_1=R_2=R_3=100$ Ohm. For optimal operation, the voltage drop across this particular LED should be 2V, while electric current through it should be I=20mA=0.02A. Find the value of resistance R_4 needed to achieve this optimal regime.



Hint: Assume that the voltage drop across LED and the current flowing through it are optimal: 2V and 0.02A respectively. Let current through resistor R_1 be I_1. From here, find currents and voltages across other resistors.

Answer: 20 Ohm

Solution: Assume that the voltage drop across LED and the current flowing through it are optimal: 2V and 0.02A respectively. Let current through resistor R_1 be I_1, then the current in R_2 is I_1-0.02A. The total voltage drop across R_1 and R_2 Is 10V, so $10V = 100\Omega\,(2I_1 - 0.02A)$. By solving this equation, we obtain $I_1 = 0.06\,A$. This means that Voltage drop across R_3 is $V_3 = I_1 R_1 + 2V = 8V$, and the current $I_3 = 8V/100\Omega = 0.08A$.

We can now calculate both voltage drop and current across the unknown resistor R_4: $V_4 = 10V = V_3 = 2V$; $I_4 = I_3 + 0.02A = 0.1A$. Hence, $R_4 = V_4/I_4 = 20\ Ohm$.

CHEMISTRY

5 points:

"*We had two bags of Arabica coffee, seventy-five ounces of sodium hydroxide pellets, five kilograms of high purity acetic acid, a saltshaker half-full of zinc chloride, and a whole galaxy of multi-colored pH papers, rubber balloons, strings etc... Also, a quart of concentrated hydrochloric acid, a quart of acetone, a case of Poland Spring water, a pint of raw ether, and two dozen grams of some amyl alcohol. Not that we needed all that for our trip, but once you get locked into a serious chemicals collection*, *the tendency is to push it as far as you can. I even was not sure which amyl alcohol did we have: iso-amyl, sec-amyl, tert-amyl (a friend of mine, Lucas, explained me that he'd just invented a simple test to discriminate them), and I was absolutely confident we would be quite capable of figuring that out. The only thing that really worried me was the ether. There is nothing in the world more helpless and irresponsible and depraved than a man in the depths of an ether binge, and I knew we'd get into that rotten stuff pretty soon.*"

Am I right, and will I (using the chemicals from our list and with Lucas help) be capable of figuring out which of three alcohols do we have? If yes, describe the procedure.

Hint:

Actually, the hint is already there. We really can do that with Lucas help.

Solution:

Primary, secondary and tertiary alcohols differ in the number of alkyl substituents attached to the carbon atom adjacent to the alcohol's OH group. Thus, if we take methanol (CH_3-OH) as the simplest alcohol (which is true), ethyl alcohol (CH_3-CH_2-OH) is a primary alcohol (only one carbon is attached to the CH_3 group), whereas tert-butyl alcohol is a tertiary alcohol (three alkyl groups are attached to this carbon). You can read the Wikipedia article for more details, it currently contains no significant errors.

The chemical reaction that discriminates between primary, secondary and tertiary alcohols is called the Lucas' test. It is based on the differences in reactivity of the OH group in primary, secondary and tertiary alcohols towards acids, and HCl in particular. This reaction is an exchange (a.k.a. "double replacement" reaction) where the alcohol and the acid produce the halogenoalkane (a.k.a. alkyl halogenide) and water:

$$R\text{-}OH + HCl \rightarrow R\text{-}Cl + H_2O$$

In this equation, "R" denotes the rest of the alcohol's molecule (usually, an alkyl residue). The product of this reaction, alkyl halogenide (R-Cl), is much less soluble in water or HCl than a

parent alcohol, which means its formation is easy to see (the reaction mixture becomes turbid). Zinc chloride serves as a catalyst in this reaction.
In primary alcohols, the bond between the OH group and the rest of the molecule is strong, and when the solution of zinc chloride in HCl (a.k.a. “Lucas’ reagent”) is added to it, the reaction is slow. In contrast, tertiary alcohols react rapidly. Secondary alcohols have intermediate reactivity. This fact provides us with an easy tool to discriminate between these three types of alcohols. To do the Lukas’ test, you have to add the Lucas’ reagent to your alcohol and stir the mixture. If the mixture becomes turbid immediately, and an oily product (alkyl halogenide) forms, your alcohol is a tertiary alcohol. If the reaction is slow, and any visible changes are seen only after 5-10 minutes, your alcohol is a secondary alcohol. If you see no reaction at room temperature, and some heating is required to see any changes, your alcohol is a primary alcohol.

10 points:

Alice, a college faculty, came to her lab and found Bob, her technician, smiling meditatively. “*I know that look*”, Alice said. “Were you binge-watching YouTube videos again?” *Yes!*”, Bob said. “*People throw one-pound pieces of sodium into ponds, and it is amazing! We should do it right now. I found enough sodium in the lab; let’s do it together – it is going to be awesome!*” “*Wait*”, Alice said, “*Do you want to throw a pound of sodium into a pond in front of our building?*” “*Yes! YES!!*” “*But what about all the fish? Freshwater fish do not tolerate water when its pH is above 9. If throwing one pound of sodium into the pond will increase the pH of water up to 9 or higher, I would say you should not do that*”, Alice said. "*I recall the average depth of the pond is 2 meters and its diameter is 40 meters. Can you please calculate the pH of water in the pond after throwing 1 pound of sodium there, and depending on the results, we will decide if we can do this experiment. In you calculations, assume the pH is exactly 7 now, and the water is fresh and free of other dissolved substances.* "

Make needed calculations and tell if Bob will be allowed to do this experiment.

Hint:

Just calculate the concentration of OH ions assuming that each sodium atom produces one OH^- upon reaction with water. A negative logarithm of concentration of OH^- is pOH, and pH+pOH always equal to 14.

Solution:

pH of some solution is a negative decimal (common) logarithm of concentration of hydrogen ions:

$pH = -lg[H^+]$

Square brackets mean that we are talking about actual concentration of some species, and “lg” is common logarithm. Logarithm is used because $[H^+]$ varies very significantly, from 1 M or even less (in strong acids) to 10^{-14} M (in strong bases), so, instead of writing “10^{-5}” it is more

convenient to write “-5”. A negative sign is needed to make our life even more easy: when $[H^+]$ in some solution is 10^{-5}, its pH is just 5, which is very convenient.
The second important fact is that the product of concentrations of H^+ and OH^- is *always* the same is *all aqueous solutions*. This product is equal to 10^{-14}. That means
pH + pOH = 14.
(obviously, pOH is defined in the same way as pH; pOH is used rarely, because it is redundant, however, in this problem, we will use it, but only once.)
From all said above, we conclude the solution of this problem has three steps.

1. Calculate the concentration of hydroxide ions ($[OH^-]$) in the pond when one pound of sodium has been thrown there.
2. Calculate pOH (by analogy with pH, pOH = -lg[OH-]).
3. Calculate pH using the formula pH + pOH = 14.

Let’s start.

1. First of all, let’s calculate the pond’s volume. Its radius is 20 m, which means its area is $3.14\times20^2=1256$, and the volume is 2512 m^3, or 2512000 L.
   In the reaction between sodium and water:

   $2\ Na + 2H_2O \rightarrow 2NaOH + H_2$

   one OH^- ion is produced by one Na atom, which means, the number of moles of sodium reacted equals to the number of moles of hydroxide produced. One pound of sodium is approximately 450 grams, or 450/23=19.57 mols. That means by throwing 19.57 mols of sodium in 2512000 L of water, we create a NaOH solution with $19.57/2512000 = 7.7\times10^{-6}$ M concentration. We know that NaOH is a strong electrolyte, which means it completely falls apart in aqueous solution, so each NaOH molecule produce one OH^- ion, so the concentration of OH^- ions will be 7.7×10^{-6} M.
2. That allows us to calculate pOH, which is:

   $pOH = -lg(7.7\times10^{-6})=5.1.$

3. Now we are ready to calculate pH

   pH = 14 - pOH = 14 - 5.1 = 8.89.

Hurrah! pH value is less than 9, so Alice’s requirement is met, and Bob IS allowed to do this experiment!
Actually, although the calculated pH value is pretty close to 9, there will hardly any harm to environment, because in real life, pH of fresh water is lower than 7 due to absorption of carbon dioxide from atmosphere. In addition, due to the presence of dissolved salts in natural water (as a rule, they are calcium and magnesium bicarbonates), the water in ponds and rivers changes its pH less significantly than chemically pure water. Due to these two reasons, the actual pH in the actual pond will be even lower than 8.89, which means this experiment is totally environment friendly.

BIOLOGY

5 points:

Based on what observations scientists might have assumed that the extinct species of animals was warm-blooded?

Answer: Modern animals are often characterized as "warm-blooded" (mammals, birds) and "cold-blooded" (everything else). This is a simplification of several related phenomena: energy source (endothermy vs. ectothermy); metabolic rate (tachymetabolism vs. bradymetabolism); and temperature stability (homeothermy vs. poikilothermy).
Birds and mammals are warm-blooded; that is, they are warmer than the environment around them in typical temperate and colder environments. Crocodilians, lepidosaurs, turtles, amphibians, most fish, and almost all invertebrates are cold-blooded: their bodies are generally only about as warm as the general environment around them, so consequently they feel cool to the touch outside of tropical situations; in contrast, warm-blooded animals have temperatures largely independent of the outside temperature, so they feel warm to the touch.

"Warm-blooded" and "Cold-blooded" actually encompass several different (although related) topics:

1. Energy Source: whence comes the majority of the energy to "run" the animal?

In "Cold-blooded" animals, the main energy source is the sun (and external environments in general): called ectotherms ("outside heat") In "Warm-blooded" animals, the main energy source are specialized sub-cellular structures whose main purpose is to convert food energy to heat energy: called endotherms ("inside heat")

2. Metabolic Rate: how much food energy ("fuel") is used up over time?

In "Cold-blooded" animals, rate of fuel usage is low: called bradymetabolic ("slow metabolism") In "Warm-blooded" animals, rate of fuel usage is HIGH: called tachymetabolic ("fast metabolism")

3. Temperature Variation over Time: how stable is the body temperature over time?

In "Cold-blooded" animals, body temperature fluctuates with the external environment: called poikilotherms ("fluctuating heat"). In "Warm-blooded" animals, body temperature regulated by internal mechanisms and thus more stable: called homeotherms ("same heat")

A typical cold-blooded animal is an **ectothermic bradymetabolic poikilotherm**: needs to get its energy from the sun and fluctuates with external environment (but can moderate fluctuations by moving from sunlight to shade and vice versa); however, needs very little food (snakes can go weeks without feeding, for example). Cold blooded animals become torpid at night and in colder weather.

A typical warm-blooded animal is an **endothermic tachymetabolic homeotherm**: its body temperature is stable and activity levels can remain high for long periods of time, at night, and in colder weather; however, needs a LOT of food or will die (imagine the effects of not feeding a cat or dog for weeks...).

How can people determine the thermal physiology of extinct animals like non-avian dinosaurs?

To determine it scientists should take a lot of factors into consideration. So far there are no evidence that one single factor can provide an univocal answer.
Some factors which help scientists to conclude whether extinct animal was “warm” or “cold” blooded:

- Upright posture: today, all living animals with upright posture are warm-blooded. Problem: No causal relationship ever established: just because all living animals with upright stance are endotherms does not mean that upright stance requires endothermy
- Dental batteries of hadrosaurids and ceratopsids: useful for chopping up food into very fine particles for fast digestion
- High blood pressure necessary to pump blood into brains of tall theropods, ornithopods, and (most especially) sauropods: requires powerful, active heart
- Latitudinal distribution: dinosaurs (and therapsids) found in Mesozoic (and Permian-Triassic) polar regions, although not as cold as today would still be cooler than climates preferred by typical modern cold-blooded animals
- Bone microstructure: lots of signs of reworking (bone being resorbed as mineral source in metabolism, and redeposited), lots of Haversian canals.
- Growth rate: fantastic growth rate (see earlier lecture) suggests tachymetabolism.
- Hearts, Lungs, and Faces (endotherms have special features)

One of the most important arguments on which the assumption that extinct species might be cold or warm blooded was the fact of their bones show "growth rings" (Lines of Arrested Growth, or LAGs), typical of reptiles and (once thought to be) lacking in mammals. But in 2012 a comprehensive global study of wild ruminants from tropical to polar environments was published in Nature (https://www.nature.com/articles/nature11264), where authors showed, that cyclical growth is a universal trait of homoeothermic endotherms. Their results show that the presence of these lines is not an indicator of an ectothermic physiology (does not generate internal heat), as had previously been thought, since all warm-blooded mammals have them. The study therefore dismantles the key argument of the hypothesis that dinosaurs could have been cold-blooded reptiles.

10 points:

In the late 1980th, Carolyn Napoli has studied flower coloration in petunias. She knew that the intensity of the purple color was determined by the chalcone synthase (CHS), a key enzyme in the biosynthesis of the purple pigment, anthocyanin. Carolyn hypothesized that in order to produce darker purple flowers, she would need to engineer plants with extra copies of the CHS gene. The rationale was straightforward: the more copies of the gene, the more mRNA is produced, so the level of CHS will go up, and the color of the flowers will be deeper. However, the result was utterly unexpected - instead of the dark purple, Dr. Napoli ended up with white flowers, and, after several years of extensive studies, a biologist discovered a totally new protein machinery that is present in almost all eukaryotic organisms. What this machinery is, how does it explain the white color of genetically modified petunias, and what is the primary function of this machinery in eukaryotes?

Answer:

This controversy can be explained by the phenomenon called RNA interference, or RNAi (co-suppression, in terms of the research team).

To understand how it works, one have to keep in mind that the only source of RNA in plant or animal cells is the process called transcription, i.e. the synthesis of a single RNA strand using DNA as a template. That means RNA is produced in a single-stranded form ssRNA: no complementary strand is synthesized, and no double stranded RNA (dsRNA) can from in animal or plant cells normally. In contrast, some viruses or other parasites may have RNA as a primary storage of genetic information. In double-stranded RNA viruses, RNA is being directly copied during viral replication, and large quantities of double stranded RNA form in infected cells. Therefore, the presence of long segments of dsRNA in cells is an alarm signal indicating that the cell has been invaded some parasite. This alarm signal triggers the cellular response, and this response has two steps. First, a special protein complex, Dicer, binds to the dsRNA and chops it by short segments, 19-21 nucleotide long. By doing that, Dicer destroys invader's genome, but it does not destroy the RNA that may exist in a single stranded form and serve as a template for protein synthesis and/or RNA replication of the virus. Therefore, the second protein complex, Argonaut, binds to the segments produced by Dicer and uses them as a probe to recognize the viral ssRNA: either one or another strand of the Argonaut-bound RNA segment binds to the complementary sequence in the ssRNA, and Argonaut cuts it. Since 21 nucleotide long "words" are pretty unique, and it is very unlikely that the same sequence may appear in the host's RNA, no cellular RNA can be cut by Argonaut. That is how the Dicer/Argonaut system (a.k.a. RISC) works.

Interestingly, if a double stranded RNA forms in a cell accidentally, or it is introduced there by a researcher, that triggers the same response. If this RNA has a sequence that is identical to the sequence of some cell's own gene, the RISC system will start eating the mRNA that corresponds to this gene, and production of the protein this gene encodes will stop. This effect is currently used widely by researchers (who want to understand which role some particular gene plays in cells) and in therapy of some diseases (cancer therapy that utilizes RISC is currently being developed, although it has not been approved yet).

The story described in this problem gave a start to the chain of events that eventually lead to discovery of RISC. When producing plants with more CHS, Dr. Napoli has introduced an extra copy of the desired gene, and some segments in this gene were self-complementary. During its transcription, a small number of double-stranded RNA molecules was produced leading to degradation of the CHS mRNA, and thus, white color of the plants in the experiment.

COMPUTER SCIENCE

- You can write and compile your code here: http://www.tutorialspoint.com/codingground.htm
- Your program should be written in Java or Python
- No GUI should be used in your program: eg., easygui in Python. All problems in POM require only text input and output. GUI usage complicates solution validation, for which we are also using *codingground* site. Solutions with GUI will have points deducted or won't receive any points at all.
- Please make sure that the code compiles and runs on http://www.tutorialspoint.com/codingground.htm before submitting it.
- Any input data specified in the problem should be supplied as user input, not hard-coded into the text of the program.
- Submit the problem in a plain text file, such as .txt, .dat, etc. **No .pdf, .doc, .docx, etc!**

Intro:

SigmaCamp always has a lot of returning campers and with that a lot of friends from previous years. Your program will receive on input a list of pairs of friends. For example:
Alice-Bob
Bob-Chris
Dan-Frank

5 points:

Given the list of friendships, your program needs to identify the disconnected friendship clusters. In the example above, Cluster 1: Alice-Bob-Chris, and Cluster 2: Dan-Frank. Clusters should be printed on output.

Hint:

Using *sets* makes the task much easier. Alternatively, you can keep a list of friends for each camper, and in the end combine the lists into clusters.

Solution:

Python:

```
m = int(input("enter number of friendship pairs: ").strip())
lines = []
for i in range(m):
  line = input("enter a friendship pair, e.g. Alice-Bob: ").rstrip("\r\n")
  lines.append(line)

# Let's present the friendship connections as a graph. There are 2 main presentations of graphs: adjacency list and adjacency matrix.
# Usually for sparse graphs, when E << V^2, where E is the number of edges and V is the number of vertices,
# the adjacency list is a more economical way of presentation.

adj = {} # key: person, value: list of his/her friends
name2cluster = {} # strictly speaking, it does not have to be a dictionary, we never use its value, so it could be a set
               # but let's keep so, just to have the full picture of all the labels
cluster2name = {}
cluster_label = 1
for line in lines:
  guys = list(map(str.strip, line.split('-')))
  if len(guys) == 2:
       if guys[0] not in adj:
       adj[guys[0]] = [guys[1]]
       else:
       adj[guys[0]].append(guys[1])
       # we have undirected graph because Alice -> Bob is the same as Bob -> Alice
       # add both directions
       if guys[1] not in adj:
       adj[guys[1]] = [guys[0]]
       else:
       adj[guys[1]].append(guys[0])
  elif len(guys) == 1:
       if guys[0] in name2cluster:
       raise Exception("splitting personality for %s" % guys[0])
       else:
       cluster2name[cluster_label] = [guys[0]]
       name2cluster[guys[0]] = cluster_label
       cluster_label += 1
  else:
       raise Exception("invalid input: '%s'" % line)
print(adj)

# for each guy
#   mark all his/her friends with a label (cluster number)
#   when done, create a new label and move to a new guy
for guy in adj.keys():
  to_visit = [guy]
  added = False
  while len(to_visit) > 0:
       x = to_visit.pop(0)
       if x not in name2cluster:
       name2cluster[x] = cluster_label
       if cluster_label not in cluster2name:
       cluster2name[cluster_label] = [x]
       else:
       cluster2name[cluster_label].append(x)
```

```
        to_visit.extend(adj[x])
        added = True
  if added:
        cluster_label += 1
print(cluster2name)
print("end.")
```

Java:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;

/*
Let's present the friendship connections as a graph. There are 2 main presentations of graphs:
adjacency list and adjacency matrix.
Usually for sparse graphs, when E << V^2, where E is the number of edges and V is the number of
vertices,
the adjacency list is a more economical way of presentation.
*/
public class Friends5 {
  String[] lines = {"Alice-Bob", "Bob-Chris", "Dan-Frank", "Ed"};
   Map<String, List<String>> adj = new HashMap<>(); // key: person, value: list of his/her
friends

  void input() throws IOException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("enter number of friendship pairs: ");
    String line = reader.readLine().trim();
    int m = Integer.valueOf(line);
    lines = new String[m];
    for(int i=0; i<m; i++) {
        System.out.println("enter a friendship pair, e.g. Alice-Bob, or just a single person,
e.g. Chris: ");
      line = reader.readLine().trim();
      lines[i] = line;
    }
  }

  void findDisconnected() {
    Map<String, Integer> name2cluster = new HashMap<>();
    Map<Integer, List<String>> cluster2name = new HashMap<>();
    int clusterLabel = 1;

    for(String line : lines) {
      String[] guys = line.trim().split("\\s*-\\s*");
      if(guys.length == 1) { // a lonely guy
        if(name2cluster.containsKey(guys[0]))
              throw new IllegalArgumentException(String.format("splitting personality for %s",
guys[0]));
        else {
          List<String> list = new LinkedList<>();
          list.add(guys[0]);
          cluster2name.put(clusterLabel, list);
          name2cluster.put(guys[0], clusterLabel);
          clusterLabel++;
```

```
        }
      }
      else if(guys.length == 2) {
        if(!adj.containsKey(guys[0])) {
          List<String> list = new LinkedList<>();
          list.add(guys[1]);
          adj.put(guys[0], list);
        }
        else
          adj.get(guys[0]).add(guys[1]);
        // we have undirected graph because Alice -> Bob is the same as Bob -> Alice
        // add both directions
        if(!adj.containsKey(guys[1])) {
          List<String> list = new LinkedList<>();
          list.add(guys[0]);
          adj.put(guys[1], list);
        }
        else
          adj.get(guys[1]).add(guys[0]);
      }
      else
         throw new IllegalArgumentException(String.format("neither a pair nor a single: '%s'",
line));
    }
    System.out.println("adjacency list (for pairs):");
    for(Map.Entry<String, List<String>> entry : adj.entrySet()) {
      String guy1 = entry.getKey();
      System.out.print(guy1 + " -> ");
      for(String guy2 : entry.getValue())
        System.out.print(guy2 + ", ");
      System.out.println();
    }

    // for each guy
    //   mark all his/her friends with a label (cluster number)
    //   when done, create a new label and move to a new guy
    for(String guy : adj.keySet()) {
      LinkedList<String> toVisit = new LinkedList<>();
      toVisit.add(guy);
      boolean added = false;
      while(!toVisit.isEmpty()) {
        String x = toVisit.pop();
        if(!name2cluster.containsKey(x)) {
          name2cluster.put(x, clusterLabel);
          if(!cluster2name.containsKey(clusterLabel)) {
            List<String> list = new LinkedList<>();
            list.add(x);
            cluster2name.put(clusterLabel, list);
          }
          else
            cluster2name.get(clusterLabel).add(x);
          toVisit.addAll(adj.get(x));
          added = true;
        }
      }
      if(added)
        clusterLabel++;
```

```
    }
    System.out.println("clusters:");
    for(Map.Entry<Integer, List<String>> entry : cluster2name.entrySet()) {
      int cluster = entry.getKey();
      System.out.print(cluster + " -> ");
      for(String guy2 : entry.getValue())
        System.out.print(guy2 + ", ");
      System.out.println();
    }
  }

  public static void main(String[] args) throws Exception {
    Friends5 friends5 = new Friends5();
    friends5.input();
    friends5.findDisconnected();
    System.out.println("end.");
  }
}
```

10 points:

Given the list of friendships, your program needs to identify a minimum set of new friendships that need to be established so that everyone in Sigma is at most 2 degrees of friendship separation from everyone else (that is everyone is either a friend or a friend of a friend or a friend of a friend of a friend). In the example above, either Bob-Dan or Bob-Frank are correct answers.
Friendships that need to be established should be printed on output. If no new friendships are required, your program should state so.

Hint:

A natural way to represent relationship between campers is via a graph. We could think of each camper as a graph vertex, and friendship between two campers as an edge between corresponding vertices.

Moreover, one of the most convenient ways to represent edges in a graph is via a so called adjacency matrix, which is a square matrix where 1 in a cell (i, j) denotes a presence of an edge between vertices i and j, while 0 represents an absence of such an edge. Now you can turn the task into a matrix manipulation one.

IMPORTANT NOTE: we are not asking your program to prove that the resulting set of friendships is an absolute minimum. Instead your program should attempt to minimize the number of additional connections. As we feel that the original problem statement was not clear on this, we will allow submissions of the solutions up to January 10 without deadline penalties.

Solution:

Python:

```
"""
Unlike the previous problem, here we'll use the adjacency matrix. It has a nice property:
The n-th power of the adjacency matrix's ij element, i.e. (A^n)_ij, represents the number of ways you can go
from vertex i to vertex j in the original graph.
You can read more about it here:
http://www.utdallas.edu/~jwz120030/Teaching/PastCoursesUMBC/M221HS06/ProjectFiles/Adjacency.pdf
or watch here:
https://www.youtube.com/watch?v=c4RZ4rQIgc4
Explanations of the terms can be found here:
https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms
"""

"""
The n-th degree of the adjacency matrix gives us all the walks of distance n. In our problem we are looking for
distances of 3 or less, therefore we'll add the shorter distances as well, i.e. if A is our adjacency matrix then
A + A*A + A*A*A gives all walks <= 3. See picture at the bottom.
If in this resultant matrix we see any elements of 0 it means that the vertex i cannot reach the vertex j in 3 or
less moves, so we need to befriend with somebody from this chain.
Another interesting observation we can make is that the elements on the main diagonal of A*A give you the number
of incident edges from a vertex, or, in other words, how many connections a node has. For example, in the graph A-B-C
the walk of 2 from A to A is like A->B->A. Essentially, the walk of 2 from a node to the same node is to go its
neighbor and bounce back.
So, when we've got a node that cannot be reached within 3 walks we will link it to the most connected node as defined
by A*A in the hope to become connected to most people in order to minimize the number of added connections. Please,
note that we do not claim that this will yield the minimum number of additional connections.
We will add only 1 connection at a time. This won't be the most efficient algorithm but it's great to see
how the graph evolves!
"""

import numpy as np

m = int(input("enter number of friendship pairs: ").strip())
lines = []
for i in range(m):
  line = input("enter a friendship pair, e.g. Alice-Bob, or just a single person, e.g. Chris: ").rstrip("\r\n")
  lines.append(line)


# let's count them all, so we would know the matrix dimensions
def map_names(lines):
  persons = set()
  singles = set()
```

```
  pair_lines = []
  for line in lines:
       guys = list(map(str.strip, line.split('-')))
       if len(guys) == 1:
       if guys[0] in singles: raise(Exception("splitting personality for %s" % guys))
       singles.add(guys[0])
       elif len(guys) == 2:
       persons |= set(guys)
       pair_lines.append("%s-%s" % (guys[0], guys[1]))
       else:
       raise(Exception("invalid input: '%s'" % str))

  # names map
  name2index = dict()
  index2name = dict()
  for i, person in enumerate(sorted(persons)):
       name2index[person] = i
       index2name[i] = person
  n = len(name2index)
  return pair_lines, persons, singles, n, name2index, index2name

# our connections matrix (1 is connected, 0 - no friendship)
# because our graph is not directional, i.e. Alice -> Bob is the same as Bob -> Alice
# then our matrix is symmetrical. Note that the main diagonal is always 0 (no self loops).
def build_adjacency_matrix(lines, name2index):
  A = np.zeros([n, n], dtype=int)
  for line in lines:
       guys = list(map(str.strip, line.split('-')))
       A[name2index[guys[0]], name2index[guys[1]]] = 1
       A[name2index[guys[1]], name2index[guys[0]]] = 1
  print("A =\n", A)
  return A

# find the first most connected guy
def get_most_connected_guy(A2, index2name):
  max_ind = np.argmax(np.diag(A2))
  print("%s is one of the most connected guys" % index2name[max_ind])
  return max_ind


pair_lines, persons, singles, n, name2index, index2name = map_names(lines)
A = build_adjacency_matrix(pair_lines, name2index)
A2 = A.dot(A)
print("A^2 =\n", A2)

# this is a special case when we have disconnected persons; let's connect them to the most
connected guy
if len(singles) > 0:
  max_ind = get_most_connected_guy(A2, index2name)
  for y in singles:
       print("augmenting the input with the connection: %s - %s" % (y, index2name[max_ind]))
       pair_lines.append("%s-%s" % (y, index2name[max_ind]))

  pair_lines, persons, singles, n, name2index, index2name = map_names(pair_lines)
  assert(len(singles) == 0)
  A = build_adjacency_matrix(pair_lines, name2index)
  A2 = A.dot(A)
```

```
  print("A^2 =\n", A2)

first = True
while True:
  # calc all walks <= 3
  A2 = A.dot(A)
  print("A^2 =\n", A2)
  B = A + A2 + A2.dot(A)
  print("B =\n", B)

  # see if any guys cannot be reached
  missing = list(zip(np.where(B == 0)[0], np.where(B == 0)[1]))
  if len(missing) == 0: # we're done, we've reached complete harmony
       if first: print("no new friendships are required")
       break
  first = False
  # just take the 1st one
  x, y = missing[0]
  print("%s cannot reach %s" %(index2name[x], index2name[y]))

  max_ind = get_most_connected_guy(A2, index2name)
  if y == max_ind or A[y,max_ind] > 0:
       y = x
  print("connecting %s to %s" % (index2name[y], index2name[max_ind]))
  A[y, max_ind] = 1
  A[max_ind, y] = 1
  print("A =\n", A)
  print("\n\n")

print("end.")
```

Java:

```
/*
Unlike the previous problem, here we'll use the adjacency matrix. It has a nice property:
The n-th power of the adjacency matrix's ij element, i.e. (A^n)_ij, represents the number of ways you can go
from vertex i to vertex j in the original graph.
You can read more about it here:
http://www.utdallas.edu/~jwz120030/Teaching/PastCoursesUMBC/M221HS06/ProjectFiles/Adjacency.pdf
or watch here:
https://www.youtube.com/watch?v=c4RZ4rQIgc4
Explanations of the terms can be found here:
https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms

The n-th degree of the adjacency matrix gives us all the walks of distance n. In our problem we are looking for
distances of 3 or less, therefore we'll add the shorter distances as well, i.e. if A is our adjacency matrix then
A + A*A + A*A*A gives all walks <= 3.  See picture at the bottom.
If in this resultant matrix we see any elements of 0 it means that the vertex i cannot reach the vertex j in 3 or
less moves, so we need to befriend with somebody from this chain.
Another interesting observation we can make is that the elements on the main diagonal of A*A give you the number
of incident edges from a vertex, or, in other words, how many connections a node has. For example, in the graph A-B-C
```

the walk of 2 from A to A is like A->B->A. Essentially, the walk of 2 from a node to the same node is to go its
neighbor and bounce back.
So, when we've got a node that cannot be reached within 3 walks we will link it to the most connected node as defined
by A*A in the hope to become connected to most people in order to minimize the number of added connections. Please,
note that we do not claim that this will yield the minimum number of additional connections.
We will add only 1 connection at a time. This won't be the most efficient algorithm but it's great to see
how the graph evolves!
*/

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;
import java.util.function.IntPredicate;

// There are plenty of (better) libraries for linear algebra. But let's craft something minimalistic for educational
// purposes (therefore no attempts are made to optimize or generify this).
// This handles only square integer matrices.
class Matrix {
  int[][] A;
  int n; // dimensions

  Matrix(int n) {
        this.n = n;
        this.A = new int[n][n];
  }

  void zeros() {
        for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
        A[i][j] = 0;
  }

  // unchecked
  int get(int i, int j) {
        return A[i][j];
  }

  // unchecked
  void set(int i, int j, int value) {
        A[i][j] = value;
  }

  int[] diag() {
        int[] d = new int[n];
        for(int i=0; i<n; i++)
        d[i] = A[i][i];
        return d;
  }

  Matrix multiply(Matrix B) {
        Matrix C = new Matrix(n);
```

```
        for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
        int Cij = 0;
        for(int k=0; k<n; k++)
        Cij += A[i][k] * B.get(k,j);
        C.set(i, j, Cij);
        }
        }
        return C;
  }

  Matrix add(Matrix B) {
        Matrix C = new Matrix(n);
        for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
        C.set(i, j, A[i][j] + B.get(i,j));
        }
        }
        return C;
  }

  /**
   * find first occurrence satisfying the predicate
   * @return the indices {i, j} as an array; negative if nothing was found
   */
  int[] findFirst(IntPredicate p) {
        int[] indices = {-1, -1};
        for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
        if(p.test(A[i][j])) {
        indices[0] = i;
        indices[1] = j;
        return indices;
        }
        }
        }
        return indices;
  }

  void print(String preamble) {
        System.out.println(preamble);
        for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++)
        System.out.print(String.format("%s%d", j==0 ? "" : ", ", A[i][j]));
        System.out.println();
        }
  }
}

public class Friends10 {
  String[] lines = {"Alice-Bob", "Bob-Chris", "Dan-Frank", "Ed"};
   //String[] lines = {"0-2", "1-2", "2-3", "3-4", "4-5", "6", "7", "8-9", "9-10", "10-11",
"10-12"};
  List<String> pairLines = new LinkedList<>(); // input lines with pairs of friends only
  Set<String> persons;
  Set<String> singles;
  Map<String, Integer> name2index;
```

```
  Map<Integer, String> index2name;
  Matrix A; // adjacency matrix
  int n; // its dimension

  void input() throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("enter number of friendship pairs: ");
        String line = reader.readLine().trim();
        int m = Integer.valueOf(line);
        lines = new String[m];
        for(int i=0; i<m; i++) {
        System.out.println("enter a friendship pair, e.g. Alice-Bob, or just a single person,
e.g. Chris: ");
        line = reader.readLine().trim();
        lines[i] = line;
        }
  }

  // let's count them all, so we would know the matrix dimensions
  void mapNames(String[] lines) throws Exception {
        persons = new HashSet<>();
        singles = new HashSet<>();
        pairLines = new LinkedList<>();

        for(String line : lines) {
        String[] guys = line.trim().split("\\s*-\\s*");
        if(guys.length == 1) { // a lonely guy
        if(singles.contains(guys[0]))
        throw new Exception(String.format("splitting personality for %s", guys[0]));
        singles.add(guys[0]);
        }
        else if(guys.length == 2) {
        persons.addAll(Arrays.asList(guys));
        pairLines.add(String.format("%s-%s", guys[0], guys[1]));
        }
        else
        throw new Exception(String.format("invalid input: '%s'", line));
        }

        // names map
        name2index = new HashMap<>();
        index2name = new HashMap<>();
        List<String> sortedPersons = new ArrayList<>(persons);
        Collections.sort(sortedPersons);
        int i = 0;
        for(String person : sortedPersons) {
        name2index.put(person, i);
        index2name.put(i, person);
        i++;
        }
        n = name2index.size();
  }

  // our connections matrix (1 is connected, 0 - no friendship)
  // because our graph is not directional, i.e. Alice -> Bob is the same as Bob -> Alice
  // then our matrix is symmetrical. Note that the main diagonal is always 0 (no self loops).
  void buildAdjacencyMatrix(List<String> lines) {
```

```
        A = new Matrix(n);
        A.zeros();
        for(String line : lines) {
        String[] guys = line.trim().split("\\s*-\\s*");
        A.set(name2index.get(guys[0]), name2index.get(guys[1]), 1);
        A.set(name2index.get(guys[1]), name2index.get(guys[0]), 1);
        }
        A.print("A =");
  }

  int findMostConnectedGuy(Matrix A2) {
        int[] diag = A2.diag();
        int index = -1;
        int value = Integer.MIN_VALUE;
        for(int i=0; i<n; i++) {
        if(diag[i] > value) {
        index = i;
        value = diag[i];
        }
        }
        System.out.println(String.format("%s   is   one   of   the   most   connected   guys",
index2name.get(index)));
        return index;
  }

  void connect() throws Exception {
        mapNames(lines);
        buildAdjacencyMatrix(pairLines);
        Matrix A2 = A.multiply(A);
        A2.print("A^2 =");

        // this is a special case when we have disconnected persons; let's connect them to the
most connected guy
        if(singles.size() > 0) {
        int maxInd = findMostConnectedGuy(A2);
        for(String y : singles) {
        System.out.println(String.format("augmenting the input with the connection: %s - %s", y,
index2name.get(maxInd)));
        pairLines.add(String.format("%s-%s", y, index2name.get(maxInd)));
        }

        mapNames(pairLines.toArray(new String[pairLines.size()]));
        buildAdjacencyMatrix(pairLines);
        }

        boolean first = true;
        while(true) {
        // calc all walks <= 3
        A2 = A.multiply(A);
        A2.print("A^2 =");
        Matrix A3 = A2.multiply(A);
        Matrix B = A.add(A2).add(A3);
        B.print("B =");

        // see if any guys cannot be reached
        int[] missing = B.findFirst(x -> x == 0);
        int x = missing[0];
```

```
        int y = missing[1];
        if(x < 0) { // we're done, we've reached complete harmony
        if(first)
        System.out.println("no new friendships are required");
        break;
        }
        first = false;
        System.out.printf("%s cannot reach %s\n", index2name.get(x), index2name.get(y));

        int maxInd = findMostConnectedGuy(A2);
        if(y == maxInd || A.get(y, maxInd) > 0)
        y = x;
        System.out.printf("connecting %s to %s\n", index2name.get(y), index2name.get(maxInd));
        A.set(y, maxInd, 1);
        A.set(maxInd, y, 1);
        A.print("A =");
        System.out.println();
        }
  }

  public static void main(String[] args) throws Exception {
        Friends10 friends10 = new Friends10();
        friends10.input();
        friends10.connect();
        System.out.println("end.");
  }
}
```

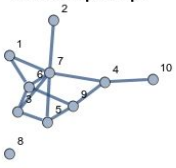
Adjacency matrix (A)
vertices 1 and 5 aren't connected

0	0	0	0	**0**	1	1	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	1	0	1	1
0	0	1	0	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0
1	1	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	0	0	0	0	0	0

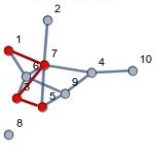
3rd power of A (# of paths of length 3 between vertices)
vertices 1 and 5 are connected by 4 such paths

2	1	3	2	**4**	6	8	0	3	1
1	0	2	0	1	2	6	0	3	1
3	2	4	4	7	9	9	0	3	1
2	0	4	0	1	2	10	0	7	3
4	1	7	1	2	3	11	0	8	2
6	2	9	2	3	4	12	0	9	2
8	6	9	10	11	12	6	0	3	0
0	0	0	0	0	0	0	0	0	0
3	3	3	7	8	9	3	0	0	0
1	1	1	3	2	2	0	0	0	0

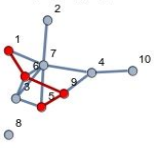
Friendship Graph



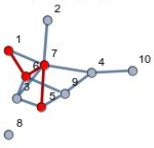
Path of length 3: 1–7–3–5



Path of length 3: 1–6–9–5



Path of length 3: 1–6–7–5



Path of length 3: 1–6–3–5