MATHEMATICS

5 points:

Imagine that you live in a 4D world. You are asked to make a wire model of the 4D cube (called hypercube) of size 10x10x10x10 cm. How much wire is needed? Wire is only used to make one-dimensional edges of the hypercube.

Hint:

Introduce a coordinate system, and find the coordinates of all vertices of the hypercube.

This will help you to count the number of edges

Answer: *320 cm*

Solution:

We need to count the number of edges. In order to do this, let us introduce the coordinate system such that axes ***x, y, z, w*** are directed along the edges of the hypercube. In this coordinate system, each of the four coordinates of a vortice ***(x,y,z,w)*** can be either 0 or 10. Each edge connects two vortices, for instance (0,0,0,0)and (0,0,0,10). in other words 3 of their coordinates are the same, and one changes from 0 to 10.

Suppose the coordinate ***w*** changes from 0 to 10. How many such edges are there? each of the coordinates ***x***,***y*** and ***z*** can be either 0 or 10, so there are $2 \times 2 \times 2$ possibilities. We just counted all edges parallel to axis ***w***. There are 4 axes in 4D space, so the final result is $4 \times 10cm \times 2 \times 2 \times 2 = 320cm$

10 points:

Imagine that you live in D dimensions. You are asked to make a wire model of a D-dimensional cube (called hypercube) of size 10x10x...x10x10 cm. How much wire is needed? Wire is only used to make one-dimensional edges of the hypercube.

Hint:

Introduce a coordinate system, and find the coordinates of all vertices of the hypercube. This will help you to count the number of edges

Answer: $10 \cdot D \cdot 2^{D-1} cm$

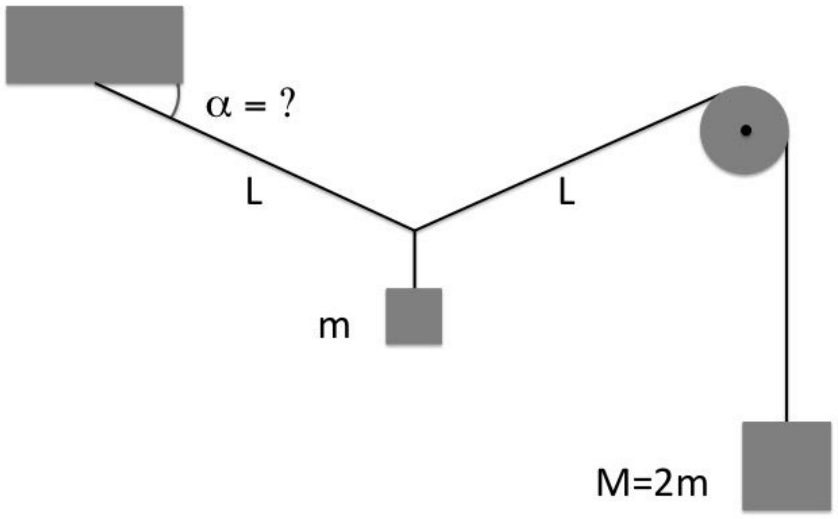
Solution:

We need to count the number of edges. In order to do this, let us introduce the coordinate system such that D axes ***x, y, z....*** are directed along the edges of the hypercube. In this coordinate system, each of the D coordinates of a vortice ***(x,y,z,....)*** can be either 0 or 10. Each edge connects two vertices, for instance (0,0,0,0,....) and (10,0,0,0,....). in other words, D-1 of the coordinates stay the same, and one changes from 0 to 10.
Suppose the coordinate ***x*** changes from 0 to 10. How many such edges are there? Each of all other coordinates can be either 0 or 10, so there are 2^{D-1} possibilities. We just counted all edges parallel to axis ***x***. There are D axes, so the final result is $10 \cdot D \cdot 2^{D-1} cm$

PHYSICS

5 points

Two weights of masses m and M=2m are attached to an ideal weightless string. One end of the string is attached to a ceiling while the other one goes around the ideal weightless pulley. Find the angle α between the string and the horizontal line in equilibrium (see figure).



Hint: The sum of forces acting on each weight should be equal to zero in equilibrium. The tension of the rope on both sides of the pulley is the same.

Answer: $\alpha = 14.5 deg = 0.25 rad$

Solution: The tension of the rope is $T = 2mg$. The total vertical projection of the tension force (from two pieces of the rope) acting on m is equal to $2T \sin\alpha = mg$. We obtain $\sin\alpha = 1/4$ and $\alpha = 14.5 deg$ or $\alpha = 0.25 rad$.

10 points

A lawn sprinkler spraying water oscillates at a constant rate up-down, from 15 to 75 degrees to the horizon. The water's exit velocity from the sprinkler is 10m/s. What fraction of water will be wasted by spraying it on a wide road whose curb is at the distance 9m from the sprinkler. The road is running perpendicular to the direction of the sprinkler.

Hint: Use the range of a projectile formula to figure out which angles correspond to "wasting water".

Answer: 43%

Solution: The range of a projectile formula tells $d = \frac{v^2}{g} sin(2\theta) = 10\ sin(2\theta)$, where the last formula gives the distance in meters and we substituted $v = 10 m/s$ and $g \approx 10 m/s^2$. We have that the range 9m corresponds to $sin(2\theta) = 9/10$, that is $\theta = 32.1$ or $\theta = 57.9$ degrees. Therefore, the wasted water is the one emitted at angles between those corresponding to the range of angles 57.9-32.1=25.8 degrees. The total range of angles is 75-15=60 degrees and the fraction of the wasted water is 25.8/60=0.43=43%

CHEMISTRY

5 points:

Imagine you are playing the Escape the Room game. In this room, you found various things including:

- porcelain cups, glasses, and dishes;
- pair of goggles and latex gloves;
- a cotton pad;
- a horseshoe magnet;
- a sheet of dirty paper, letter size, with scrawls saying “Nothing interesting”;
- a roll of pH paper, the scale attached to the roll says it measures pH from 1.0 to 6.0;
- a book “The Bottle Imp” by Robert Louis Stevenson (it is also available online at http://gutenberg.spiegel.de/buch/the-bottle-imp-4357/1);
- a printed brochure “Making invisible ink” (also available online at http://www.rsc.org/learn-chemistry/resource/download/res00001178/cmp00002225/pdf)
- a box of matches;
- four amber glass bottles; each bottle contains some transparent odourless liquid; there are four small labels on the table near these bottles; these labels look like they were attached to these bottles, but have fallen off later; it is not possible to tell for sure which label belongs to which bottle. The labels say : “green tea + Na_2SO_4”, “NaOH”, “H_2SO_4”, “black tea + $CaCl_2$”, and “$FeCl_3$”.

You already have opened nine locks, but to exit the room you need to unlock the final lock. You need to find the code that unlocks it.
What will you do to obtain the code?

Hint: Whereas “The Bottle Imp” can hardly help you in obtaining the secret number (by writing that I do not mean it is not worth reading) the “Making invisible ink” may be really helpful. It would be also useful to remember that solutions of iron (III) salts have a yellow to brown color. If you that, and if you remember the general properties of metal salts (especially, their reactions with other salts, acids and bases) you will be able to escape the room.

Solution:

Obviously, the words “nothing interesting” written on the sheet of paper imply there *is* something interesting there, namely, a message written with some invisible ink. Depending on the type, Invisible ink may become visible either upon heating or upon chemical treatment. Since there are some chemicals in the room, it is natural to conclude some chemical treatment is necessary to make a message visible. The “Making invisible ink” brochure (page 3) says that potassium thiocyanate is one type of invisible ink that can become visible upon treatment with iron (III) chloride, or $FeCl_3$ solution (potassium thiocyanate reacts with iron (III) salts to form iron thiocyanate, which has a bright red color). There is a bottle with $FeCl_3$ solution in the room, which supports the hypothesis that potassium thiocyanate was used as a secret ink. However, we do not know which bottle contains $FeCl_3$, because all bottles have no labels. Iron (III) salts are colored (usually they have a brown to yellow color), however, there are *three* bottles with colored liquids ("green tea + Na_2SO_4", "black tea + $CaCl_2$", and "$FeCl_3$"). How can we tell which is which? To answer this question, we need to remember that most salts formed by divalent or trivalent metals (and $FeCl_3$ is a typical example) react with NaOH to form a loose precipitate of insoluble hydroxide. For $FeCl_3$, the reaction is as follows:

$$FeCl_3 + 3\ NaOH \rightarrow Fe(OH)_{3\ (solid)} + 3\ NaCl$$

In other words, when you add a NaOH solution (i.e. one of the two colorless solutions in the room) to each of the three yellowish or brownish solutions (which are supposed to be "green tea + Na_2SO_4", "black tea + $CaCl_2$", or "$FeCl_3$"), the dark and loose precipitation will occur only in the solution containing $FeCl_3$. To do that, pour a small amount of each solution to a separate cup or glass, and add few drops of the first colorless solution to each cup. If the solution you are adding is NaOH, the dark precipitate will form one of the three cups.
If no dark precipitate is formed, then the solution you took was not NaOH, but H_2SO_4 (which is colorless too). In that case you might observe formation of *white* precipitate with the "black tea + $CaCl_2$" (this precipitate is gypsum, or $CaSO_4$), but no loose and dark precipitate will form. In that case, just forget about this solution, and take the second colorless solution, which will be NaOH, and repeat your experiment again. Finally, you will identify the bottles with NaOH and $FeCl_3$ solutions. Take the latter solution, and using a cotton pad treat the sheet of paper. Red letters will become visible on it, which will give you a clue on how to escape the room.

10 points:

Alice and Bob, her technician, have been preparing for tomorrow's class. "Tomorrow, I am going to tell our students about the most common methods of purification of chemical compounds" - Alice said. "I plan to start with recrystallization, which is carried out in the following way: I'll take some compound, for example, some salt, which is contaminated with another chemical (for example, with some dye or with some other salt), and dissolve this mixture in a minimal amount of hot water. Then we will put it on ice for cooling. After some period, crystals of the purified compound will start to form and precipitate, whereas the admixtures remain in solution. Then you Bob will collect the crystals by filtration, and leave them to dry. " "Great", Bob said, "And which compounds should I prepare for tomorrow's demonstration?" "Maybe, you yourself have some ideas on that account?" - Alice said.
"Yes, I do", Bob replied. "Let's take sodium chloride and add a small amount of copper sulfate to it. The initial mixture will be bluish, but upon recrystallization we will obtain clear, transparent and colourless crystals of NaCl. That would be spectacular. Instead of sodium chloride we may take potassium sulfate (K_2SO_4) or cerium sulfate ($Ce_2(SO_4)_3$)."
"I am sorry, Bob, but, although you are thinking in a right direction, only one of your suggestions will work," Alice said.

Please tell if Alice was right. Which compounds proposed by Bob cannot be used to demonstrate the principle of recrystallization, and why?

Hint: Classical recrystallization is based on the observation that solubility of most compounds increases at elevated temperature. Is that the case for the salts Bob proposed to use during the next class?

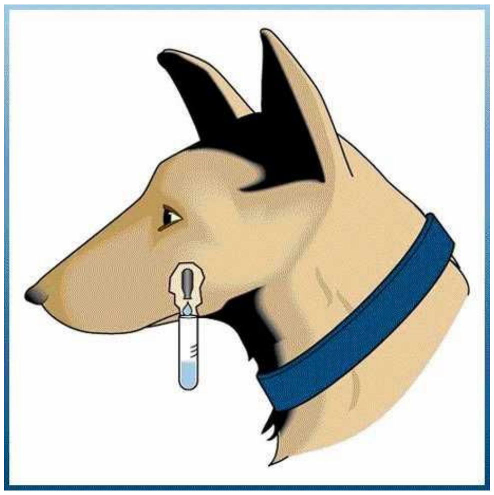
Solution:

For recrystallization to be possible, solubility of the compound that is subjected to recrystallization must be significantly greater at high temperature. That is the case for most inorganic and organic compounds, but the two salts proposed by Bob (sodium chloride and cerium sulfate) are the exception. Solubility of sodium chloride in water is almost constant at all temperatures, from zero degrees Celsius to 100 degrees. With regard to cerium sulfate, its solubility in water even decreases at elevated temperature. Out of the tree salts proposed by Bob, only potassium sulfate is suitable to demonstrate the principle of recrystallization.

BIOLOGY

5 points:

In the second half of XIX century Russian physiologist Ivan Pavlov studied digestive system in dogs. He implanted a special tube (a fistula) into dog's cheek (see a picture) and measured the amount of saliva produced by a dog in different situations. Surprisingly, he observed dogs start to salivate not only when they are eating, but also when they see a food, or even when they hear a sound indicating the start of feeding time. Based on this and other experiments, Pavlov developed a theory of conditioned reflex, for which he was awarded the Nobel prize in 1904. Interestingly, when scientists did similar experiments with cats, they observed cats do not start to salivate when they see food, or when they hear the signal indicating a food is coming soon. Usually, cats start to salivate only when they get a food. Can you explain why?



Answer:

The explanation is in different hunting strategy of cats and dogs. Whereas dogs bring down their prey through chases, cats ambush their prey. Accordingly, for dogs, the time interval between seeing their food closely and having it is relatively short. In contrast, cats may spend hours and hours sitting in an ambush, in close proximity to their potential food. Since the time between seeing, hearing, or smelling their prey and having it is long, no unconditional salivation reflex was developed in cats that forces them to salivate upon seeing food. Indeed, for cats, it would be wastefulness to salivate continuously during several hours sitting in an ambush near a mouse that is hiding in a hole just few inches from the cat. Only when cats get food they may afford to start salivating.

10 points:

In a *symbiotic* relationship, two organisms each receive benefit from living next to one another. *Endosymbiosis*, an extension of this idea, is an evolutionary theory that proposes that several of our key cell components originated as a symbiosis between separate single-celled organisms. One key example of this is mitochondria, which supply energy to our cells, and which may have originated as bacteria.

1. What are the three types of symbiosis? Give an example of each type of symbiotic relationship amongst animals, and explain why it reflects each type.
2. What evidence makes scientists think today that mitochondria may have originated as bacteria?
3. If mitochondria did originate as bacteria, would this have been a symbiotic relationship? If so, which of the three types and why?
4. How are mitochondria different from bacteria today, and what evolutionary mechanism might explain why this is so?
5. How does this relate to the recent scientific finding (July 2013) that antibiotic use damages mitochondria?

Answer:

1. Three types of symbiosis are:

Mutualism: A symbiotic relationship where both organisms benefit from the interaction.
Examples: oxpecker bird and rhinoceros, honeybees and flowers, gut bacteria and humans, spider crabs and algae, etc.
Parasitism: A symbiotic relationship where one organism benefits from the interaction, to the detriment of the other organism.
Examples: tapeworms (or other types of worms) and other animals, lice or fleas and animals, leeches and other animals, parasitic bacteria and other organisms
Commensalism: A symbiotic relationship where one organism benefits from the interaction, but the other organism is not affected.
Examples: clownfish and anemones, barnacles and whales (or other marine creatures), burrs and animal fur (or humans), flatworms and horseshoe crabs, etc.

2. The hypothesis about bacterial origin of mitochondria had been proven relatively recently, when genome of large amount of bacteria and eukaryota became available (about 10 years ago). To understand how the scientists confirmed this hypothesis, it is necessary to know that bacteria and eukaryota are two different kingdoms of life, and almost everything in their cells is different. Even those proteins that have similar functions in bacterial and eukaryotic cells have totally different aminoacid sequence (in other words, the difference between bacterial and eukaryotic proteins is like a difference between fish and whales: both of them live in water, both of them have a similar body shape, but their organisms are built totally differently). Interestingly, mitochondria have their own small genome, which encodes several proteins and ribosome RNAs, and these genes have a substantial similarity with analogous genes of some modern bacteria, and almost no similarity with the genes that encode cellular proteins. That observation

became an ultimate evidence, and now the idea about the bacterial origin of mitochondria is a firmly established fact.

The detailed analysis of many genomes of eukaryotic and bacterial organisms demonstrated that about 1 billion years ago some ancient unicellular organism (similar to modern archaea) and some ancient bacteria (similar to modern rickettsia) started to live in symbiosis, and that event gave a start to a new kingdom of life, Eukatyota. Before that, only bacteria and archaea (unicellular organisms that looked like bacteria, but that had a totally different cellular structure) existed on the Earth.

Interestingly, archaea still exist on the Earth, although they are much less abundant than bacteria.

3. A billion years ago, when bacteria and their host, ancient archaea, started to live together, their symbiosis was likely to be parasitism: a host cell encapsulated the bacteria (a parasite) in a bubble formed by an internal membrane, and the bacteria developed its own stable membrane to resist to the host cell’s attempts to kill it (this double membrane, composed of the outer layer, and the inner layer, a former bacterial membrane, is still present in all modern mitochondria). However, gradually, both the host (ancient archaea) and the parasite (ancient rickettsia) developed a way to coexist: the host cell started to provide their parasites with carbohydrates, and the bacteria started to produce ATP both for their own needs and for their host. Gradually, all ATP production function in the eukaryotic cell was delegated to the bacterial symbiont (which gradually converted to mitochondria), and the host cell provided mitochondria with food and protection.

In summary, the relationship between mitochondria and the eukaryotic cell is an extreme example of mutualism: both mitochondria and their cell are absolutely necessary for each other’s survival, and they are incapable of existing separately from each other.

4. Since the bacteria, which gradually evolved into mitochondria, is not a free living organism any more, majority of its genome became redundant, and its size decreased dramatically (about 1000-fold), so only the most essential genes are preserved in the former bacterial (now mitochondrial) genome. Moreover, a part of genes moved from mitochondria to the nucleus, so currently most eukaryotic cells produce up to 90% of mitochondrial proteins in cytoplasm, and only after that these proteins are transferred to mitochondria. That is needed because ATP production (the major process that occurs in mitochondria) creates a very aggressive media that is harmful to mitochondrion’s own DNA, so the less number of genes are in mitochondria, the better.

5. As we already know, most bacterial proteins have a structure that is totally different from the structure of eukaryotic proteins that have the same function. That is why antibiotics are so efficient: they target some bacterial enzyme and deactivate it, but they are totally harmless for our own enzymes, because their structure and mechanism of action is different. However, since most mitochondrial genes have a bacterial origin, some recently developed antibiotics that “kill” bacterial proteins may also deactivate their mitochondrial analogs. That inhibits ATP synthesis in mitochondria, and the cell “runs out of fuel”. That affects mostly those cells that consume the largest amount of energy (muscular and neural cells), so the main effect of these drugs is myodystrophy and neurodegenerative disorders.

COMPUTER SCIENCE

- You can write and compile your code here: http://www.tutorialspoint.com/codingground.htm
- Your program should be written in C, C++, Java, or Python
- Any input data specified in the problem should be supplied as user input, not hard-coded into the text of the program.
- Please make sure that the code compiles and runs on http://www.tutorialspoint.com/codingground.htm before submitting it.
- Submit the problem in a plain text file, such as .txt, .dat, etc.
  No .pdf, .doc, .docx, etc!

5 points:

Your friend is a bridge engineer, tasked with designing 1000 unique bridges out of ASCII characters. The following are the building blocks:

```
^
.
.
.
-
-
-       ^
- .     |       =
- .     | ^     =
- - . . | |     =
# - - * | |     X X
# - - . | | =   X *
# # - * * * = = X X
# # # * * * = = X *
a b c d e f g h i j
```

The engineer constructs the bridge from left to right by sandwiching the building blocks together, controlling how much to offset each block from the bottom. Two blocks cannot be put on top of each other.

Your friend is asking you to write him a program that would allow him to type in a string specifying how to arrange the blocks, and the program would draw the bridge. He proposes you do it as follows: Each block is denoted by its letter (a-j), followed by a number which specified the vertical offset (must be 0 or greater). Thus, the following string:

h6g6a1a0a0b2b2c3c3e5h5h5g4h5g4h5g4h5g4h5tg4h5h5e5c3c3b2b2a1a1g6h6h6h6h6h6

should generate the following output on the screen:

```
  ^^                       ^^
  ..    ^             ^    ..
  ..    |             |    ..
  ..    |             |    ..
  --    |             |    --
 =--..  |             |  ..--=
==--....|             |....--======
==------*=============*------======
  ------*=============*------
  ------   = = = = =   ------
  ######               ######
  ####                   ####
  ##                       ##
  ##                       ##
```

Please write the script to make your friend’s life easier. If you feel like it, make a random bridge generator also :)

Solution:

```
#!/usr/bin/python

# based on an input string that specifies a bridge, e.g.
# "h6g6a1a0a0b2b2c3c3e5h5h5g4h5g4h5g4h5g4h5g4h5h5e5c3c3b2b2a1a1g6h6h6h6h6h6"
# and the building blocks defined below, draw the bridge

from __future__ import print_function
import sys
import re

blocks = { 'a' : '####------...^',
           'b' : '##---..',
           'c' : '#---.',
           'd' : '**.*.',
           'e' : '**|||||^',
           'f' : '**|||^',
           'g' : '===',
           'h' : '==',
           'i' : 'XXXX===',
           'j' : '*X*X'
         }

def main():
  print("Please enter a bridge specification: ")
  # bridge = sys.stdin.readline().strip()
  bridge = "h6g6a1a0a0b2b2c3c3e5h5h5g4h5g4h5g4h5g4h5g4h5h5e5c3c3b2b2a1a1g6h6h6h6h6h6"

  # verify the input: it must be a series of a letter between a and j followed
  # by a number from the very beginning to the very end of the string
  if not re.match("^([a-j]\d+)+$", bridge):
    print("invalid input: {}".format(bridge))
    return 1

  # separate out keys to the building blocks and offsets
  ablock  = [] # array of block keys
  offsets = [] # corresponding array of offsets
  parts = re.split("([a-j]\d+)", bridge)
  for part in parts:
     if not part: # python intersperses the split parts with empty strings, so we'll ignore
them
      continue
    [ignore, key, offset] = re.split("([a-j])", part)
    ablock.extend(key)
    offsets.extend(offset)

  tallest = 0  # max(block + offset)
  heights = [] # block + offset
  for i in range(len(offsets)):
    h = len(blocks[ablock[i]]) + int(offsets[i])
    heights.append(h)
    if h > tallest:
      tallest = h

  # draw the bridge from top to bottom
  for row in range(tallest+1):
    for col in range(len(offsets)):
      if tallest-row >= heights[col]:
        print(' ', end='')
      else:
        ind = tallest - int(offsets[col]) - row
        if ind < 0:
          print(' ', end='') # offset
        else:
          print(blocks[ablock[col]][ind], end='')
    print()
  pass

if __name__ == '__main__':
  main()
  print("end.")
  sys.exit(0)
```

Output:

```
 ^                         ^^
  .^^                       ..
  ...    ^             ^    ..
  ...    |             |    ..
  -..    |             |    --
  ---    |             |    --
 =---..  |             |  ..--=
==---....|             |....--======
==-------*=============*------======
  -------*=============*------
  #------   = = = = =   ----##
  #######               ######
  #####                   ####
  ###                       ##
   ##

hgaaabbccehhghghghghghheccbbaaghhhhh
661002233555454545454555332211666666

blocks = { 'a' : '####------...^',
           'b' : '##---..',
           'c' : '#---.',
           'd' : '**.*.',
           'e' : '**|||||^',
           'f' : '**|||^',
           'g' : '===',
           'h' : '==',
           'i' : 'XXXX===',
           'j' : '*X*X'
         }
```

10 points:

One of important focuses of computer science in the real world is storing data efficiently. For example, here is a simple variant of a real-life problem:

You are tasked with developing a way to “save to disk” the game of checkers. The requirements are as follows:

1) The whole game be stored as a string of text. A string of text is effectively a 1D array of characters.

2) The array should contain information sufficient to re-play every step of the game.

Your solution will be scored by the efficiency of your storage, that is, how few characters you can use to encode an entire game (of course, it varies by length of the game). As a hint, think about how much "information" happens at every step of the game.

As a bonus, write a program to demonstrate how your data can be decoded to display every step of the game. (You could generate text output on screen by letting "." stand for an empty white space, "*" for empty black space, '1' for player 1’s regular pieces, '2' for player 2’s regular pieces, etc.., and draw the board that way).

Solution:

```
First, let's consider a couple of common notations for checkers.

1) Chess like
  https://en.wikipedia.org/wiki/Russian_draughts
  For example,
    1. e3-d4 d6-c5
    2. g3-f4?? c5:e3:g5
    3. ...

2) English draughts
  https://en.wikipedia.org/wiki/English_draughts
  For example,
    1. 9-14 23-18
    2. 14x23 27x18
    3. 5-9 26-23

Let's show that both representations are equivalent and we only need 2 characters per move.
There are only 32 reachable positions on the board. The English alphabet has only 26 letters
but in computer the upper case and lower case letters are different, so we can map:

1) A1 -> A
   C1 -> B
   E1 -> C
   ...
   C7 -> Z
   E7 -> a
   H8 -> f

   So, our example from #1 will look like
   KN,VR,
   LO,RKT,

  Because a player can make multiple moves during his/her turn (when jumps over) I have to use
a separator, say, comma, between moves of each player. Please note that I don't need to
explicitly indicate who moves next since the players always move in turns and there is a rule
for who starts first (white or black/red). I also don't need to indicate whether it was a
regular move or a jump. I can deduce it from the distance between the start and end positions
```

of a move.

```
2)  1 -> A
    2 -> B
   ...
   26 -> Z
   27 -> a
   ...
   32 -> f

   And our example from #2 will look like
   IN,WR,
   NW,aR,
   EI,ZW,
```

So, we've managed to use only 1 character per position. But let's note that a character usually takes either 1 byte (ASCII encoding) or 2 bytes for (modern) Unicode. That's 255 or 65535, correspondingly. But we have only 32 distinct positions. Let's number the positions from 0 to 31 (instead of 1 to 32), and represent 31 in binary. We get 0b11111.

To quickly remind you what it means let's consider a (regular) number 12,345. We can factor it as
1*10^4 + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0 = 10000 + 2000 + 300 + 40 + 5 = 12,345.
By the same token our binary number can be factored as powers of 2 (hence the name binary):
0b11111 = 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 1*16 + 1*8 + 1*4 + 1*2 + 1 = 31

The reason computers work in binary is because it's easier to make an electronic component - the electric current either goes through the component or not (I oversimplified just to make a point).

Therefore, for our purpose we only need 5 bits per position or 10 bits per regular move. We'll use an additional bit as a separator. For example, 0 if the next position (5 bits) belongs to the next player, and 1 if the next position belongs to the same player (in case of the 2nd jump).

However, the problem calls for a string of text. Therefore, once we encode all game positions in 5 bits units and 1 bit separator between players, we combine them into a sequence of bits. Then we convert it to characters before storing to disk. We do it in 2 steps. First we look at the sequence as chunks of 8 bits. Never mind that our 5 bits position may end up in 2 different chunks. When we read from disk we'll do this procedure in reverse, so we can restore all the chunks. As you know 8 bits represent 1 byte. Now we convert these bytes into characters. There are multiple schemes for this but the most popular is Base64 (https://en.wikipedia.org/wiki/Base64).

Now let's write a program that
1) takes a sample game shown in https://en.wikipedia.org/wiki/English_draughts
2) encodes it using our A-f scheme
3) "stores it to disk"
4) then takes this textual representation, decompresses it and replays the game back.

```
[Event "1981 World Championship Match, Game #37"]
[Black "M. Tinsley"]
[White "A. Long"]
[Result "1-0"]
1. 9-14 23-18 2. 14x23 27x18 3. 5-9 26-23 4. 12-16 30-26 5. 16-19 24x15 6. 10x19 23x16 7. 11x20
22-17 8. 7-11 18-15 9. 11x18 28-24 10. 20x27 32x5 11. 8-11 26-23 12. 4-8 25-22 13. 11-15 17-13
14. 8-11 21-17 15. 11-16 23-18 16. 15-19 17-14 17. 19-24 14-10 18. 6x15 18x11 19. 24-28 22-17
```

20. 28-32 17-14 21. 32-28 31-27 22. 16-19 27-24 23. 19-23 24-20 24. 23-26 29-25 25. 26-30 25-21
26. 30-26 14-9 27. 26-23 20-16 28. 23-18 16-12 29. 18-14 11-8 30. 28-24 8-4 31. 24-19 4-8 32.
19-16 9-6 33. 1x10 5-1 34. 10-15 1-6 35. 2x9 13x6 36. 16-11 8-4 37. 15-18 6-1 38. 18-22 1-6 39.
22-26 6-1 40. 26-30 1-6 41. 30-26 6-1 42. 26-22 1-6 43. 22-18 6-1 44. 14-9 1-5 45. 9-6 21-17
46. 18-22 BW (Black Wins)

We leave the bits packing exercise to you ;-)
Also for simplification purpose we won't handle the cases when a piece becomes a king. You can do it by introducing a state for each piece.

Java:

```
import java.util.ArrayList;

public class Checkers {
  private char[] EnglishNotationMap = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P',
        // English draughts notation ->  1    2    3    4    5    6    7    8    9   10   11
12   13   14   15   16
                                        'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a',
'b', 'c', 'd', 'e', 'f' };
        //                          -> 17   18   19   20   21   22   23   24   25   26   27
28   29   30   31   32
  private char[][] board = new char[8][8]; // [0][0] - upper left corner

  public Checkers() {
    // initialize board
    for(int i=0; i<8; i++)
      for(int j=0; j<8; j++)
        board[i][j] = '.'; // empty white square

    // place black/red
    board[0][1] = '1';
    board[0][3] = '1';
    board[0][5] = '1';
    board[0][7] = '1';
    board[1][0] = '1';
    board[1][2] = '1';
    board[1][4] = '1';
    board[1][6] = '1';
    board[2][1] = '1';
    board[2][3] = '1';
    board[2][5] = '1';
    board[2][7] = '1';
    // empty
    board[3][0] = '*'; // black square
    board[3][2] = '*';
    board[3][4] = '*';
    board[3][6] = '*';
    board[4][1] = '*';
    board[4][3] = '*';
    board[4][5] = '*';
    board[4][7] = '*';
    // white
    board[5][0] = '2';
    board[5][2] = '2';
    board[5][4] = '2';
    board[5][6] = '2';
    board[6][1] = '2';
    board[6][3] = '2';
    board[6][5] = '2';
    board[6][7] = '2';
    board[7][0] = '2';
    board[7][2] = '2';
    board[7][4] = '2';
    board[7][6] = '2';
  }

  private int reverseMap(char ch) throws Exception {
    if(ch>='A' && ch<='Z')
      return (int)ch - 64;
    if(ch>='a' && ch<='f')
      return (int)ch - 97 + 27;
    throw new Exception("invalid input");
  }

  private int getRow(int position) {
    int row = (int) Math.ceil(position*2.0 / 8.0); // position counts only black squares, so
multiply by 2; 8 squares per row, so divide by 8
    return row-1; // make it 0 based
  }

  private int getCol(int position) {
    int row = getRow(position);
    int col = -1;
    if(row % 2 == 0) { // even
      col = position*2 % 8; // position counts only black squares, so  multiply by 2; 8 squares
per row, so division gives row, the remainder gives column in this row
      if(col == 0)
        return 7;
    }
    else { // odd row
      col = position*2 % 8 - 1;
      if(col == -1)
        return 6;
    }
    return col-1; // make it 0 based
  }

  public ArrayList<String> readInput() {
    // copy-paste from Tinsley vs Long at 1981 World Championship Match, Game #37 :
https://en.wikipedia.org/wiki/English_draughts
    String strGame = "1. 9-14 23-18 2. 14x23 27x18 3. 5-9 26-23 4. 12-16 30-26 5. 16-19 24x15
6. 10x19 23x16 7. 11x20 22-17 8. 7-11 18-15 9. 11x18 28-24 10. 20x27 32x5 11. 8-11 26-23 12.
4-8 25-22 13. 11-15 17-13 14. 8-11 21-17 15. 11-16 23-18 16. 15-19 17-14 17. 19-24 14-10 18.
6x15 18x11 19. 24-28 22-17 20. 28-32 17-14 21. 32-28 31-27 22. 16-19 27-24 23. 19-23 24-20 24.
23-26 29-25 25. 26-30 25-21 26. 30-26 14-9 27. 26-23 20-16 28. 23-18 16-12 29. 18-14 11-8 30.
28-24 8-4 31. 24-19 4-8 32. 19-16 9-6 33. 1x10 5-1 34. 10-15 1-6 35. 2x9 13x6 36. 16-11 8-4 37.
15-18 6-1 38. 18-22 1-6 39. 22-26 6-1 40. 26-30 1-6 41. 30-26 6-1 42. 26-22 1-6 43. 22-18 6-1
44. 14-9 1-5 45. 9-6 21-17 46. 18-22";
    System.out.println("   initial game: "+strGame);

    // for convenience let's split it to an array of strings
    String[] tokenGame = strGame.split("\\s*\\d+\\.\\s*"); // split by move number
    // delete empty elements
    ArrayList<String> listGame = new ArrayList<String>();
    for(int i=0; i<tokenGame.length; i++) {
      if(!tokenGame[i].isEmpty())
        listGame.add(tokenGame[i]);
    }
    return listGame;
  }

  public String compressGame(ArrayList<String> game) throws Exception {
    StringBuilder str = new StringBuilder(); // this class is faster when you need to modify
string
    for(String pairMoves : game) {
      String[] aMoves = pairMoves.split("\\s+"); // split into each opponent's moves
      if(aMoves.length==0 || aMoves.length>2)
        throw new Exception("invalid input");

      for(int i=0; i<aMoves.length; i++) { // for each opponent
        String[] moves = aMoves[i].split("[-x]");
        // TODO: verify that moves are legal
        for(int j=0; j<moves.length; j++) { // for each move
          int position = Integer.parseInt(moves[j]); // will throw NumberFormatException on
invalid input
          if(position<=0 || position>32)
            throw new Exception("invalid input");
          Character ch = EnglishNotationMap[position-1]; // use our mapping
          str.append(ch);
        }
        str.append(',');
      }
    }
    return str.toString();
  }

  public void printBoard() {
    for(int i=0; i<8; i++) {
      for(int j=0; j<8; j++) {
        System.out.printf("%c", board[i][j]);
      }
      System.out.println();
    }
    System.out.println();
  }

  /**
   * convert into standard notation and print game
   * @param text - our compressed representation
   * @throws Exception
   */
  public void displayGame(String text) throws Exception {
    printBoard();

    int moveNo = 1;
    char player = '1';
    String[] moves = text.split(",");
    for(String move : moves) {
      System.out.printf("%d) ", moveNo);
      int startRow = -1;
      int startCol = -1;
      int endRow = -1;
      int endCol = -1;
      for(Character ch : move.toCharArray()) {
        int position = reverseMap(ch);
        if(startRow < 0) {
          startRow = getRow(position);
          startCol = getCol(position);
        }
        else {
          endRow = getRow(position);
          endCol = getCol(position);
        }
        if(endRow > 0) {
          if(Math.abs(endRow-startRow)==1 && Math.abs(endCol-startCol)==1) // regular move
            System.out.print("-");
          else {                                                           // jump
            System.out.print("x");
            // remove a piece from the board
            int row = endRow - 1;
            if(endRow < startRow)
              row = endRow + 1;
            int col = endCol - 1;
            if(endCol < startCol)
              col = endCol + 1;
            board[row][col] = '*';
          }
        }
        System.out.printf("%d", position);
      }
      board[startRow][startCol] = '*';
      board[endRow][endCol] = player;
      System.out.println();
      printBoard();
      if(player == '1') // alternate
        player = '2';
      else
        player = '1';
      moveNo++;
    }
  }

  public static void main(String[] args) {
    Checkers checkers = new Checkers();
    try {
      ArrayList<String> game = checkers.readInput();
      String text = checkers.compressGame(game);
      System.out.println("compressed game: " + text); // this is what we "save to disk"
      checkers.displayGame(text);
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    System.out.println("end.");
  }
}
```

Python:

```
#!/usr/bin/python
# python implementation of checkers.java

from __future__ import print_function
import sys
import math
import re

class Checkers:
  EnglishNotationMap = [         'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P',
  # English draughts notation ->  1    2    3    4    5    6    7    8    9   10   11   12   13
14   15   16
                                 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b',
'c', 'd', 'e', 'f' ]
        #                    -> 17   18   19   20   21   22   23   24   25   26   27   28   29
30   31   32
  board = [['.' for x in range(8)] for x in range(8)] # [0][0] - upper left corner

  def __init__(self):
    # initialize board
    # place black/red
    self.board[0][1] = '1'
    self.board[0][3] = '1'
    self.board[0][5] = '1'
    self.board[0][7] = '1'
    self.board[1][0] = '1'
    self.board[1][2] = '1'
    self.board[1][4] = '1'
    self.board[1][6] = '1'
    self.board[2][1] = '1'
    self.board[2][3] = '1'
    self.board[2][5] = '1'
    self.board[2][7] = '1'
    # empty
    self.board[3][0] = '*'; # black square
    self.board[3][2] = '*'
    self.board[3][4] = '*'
    self.board[3][6] = '*'
    self.board[4][1] = '*'
    self.board[4][3] = '*'
    self.board[4][5] = '*'
    self.board[4][7] = '*'
    # white
    self.board[5][0] = '2'
    self.board[5][2] = '2'
    self.board[5][4] = '2'
    self.board[5][6] = '2'
    self.board[6][1] = '2'
    self.board[6][3] = '2'
    self.board[6][5] = '2'
    self.board[6][7] = '2'
    self.board[7][0] = '2'
    self.board[7][2] = '2'
    self.board[7][4] = '2'
    self.board[7][6] = '2'

  def reverseMap(self, ch):
    if ch>='A' and ch<='Z':
      return ord(ch) - 64
    if ch>='a' and ch<='f':
      return ord(ch) - 97 + 27
    raise Exception("invalid input")

  def getRow(self, position):
    row = int(math.ceil(position*2.0 / 8.0)) # position counts only black squares, so  multiply
by 2; 8 squares per row, so divide by 8
    return row-1 # make it 0 based

  def getCol(self, position):
    row = self.getRow(position)
    col = -1;
    if(row % 2 == 0): # even
      col = position*2 % 8 # position counts only black squares, so  multiply by 2; 8 squares
per row, so division gives row, the remainder gives column in this row
      if col == 0:
        return 7
    else: # odd row
      col = position*2 % 8 - 1
      if col == -1:
        return 6
    return col-1 # make it 0 based

  def readInput(self):
    # copy-paste from Tinsley vs Long at 1981 World Championship Match, Game #37 :
https://en.wikipedia.org/wiki/English_draughts
    strGame = "1. 9-14 23-18 2. 14x23 27x18 3. 5-9 26-23 4. 12-16 30-26 5. 16-19 24x15 6. 10x19
23x16 7. 11x20 22-17 8. 7-11 18-15 9. 11x18 28-24 10. 20x27 32x5 11. 8-11 26-23 12. 4-8 25-22
13. 11-15 17-13 14. 8-11 21-17 15. 11-16 23-18 16. 15-19 17-14 17. 19-24 14-10 18. 6x15 18x11
19. 24-28 22-17 20. 28-32 17-14 21. 32-28 31-27 22. 16-19 27-24 23. 19-23 24-20 24. 23-26 29-25
25. 26-30 25-21 26. 30-26 14-9 27. 26-23 20-16 28. 23-18 16-12 29. 18-14 11-8 30. 28-24 8-4 31.
24-19 4-8 32. 19-16 9-6 33. 1x10 5-1 34. 10-15 1-6 35. 2x9 13x6 36. 16-11 8-4 37. 15-18 6-1 38.
18-22 1-6 39. 22-26 6-1 40. 26-30 1-6 41. 30-26 6-1 42. 26-22 1-6 43. 22-18 6-1 44. 14-9 1-5
45. 9-6 21-17 46. 18-22"
    print("   initial game: "+strGame)

    # for convenience let's split it to an array of strings
    tokenGame = re.split("\\s*\\d+\\.\\s*", strGame) # split by move number
    # delete empty elements
    if not tokenGame[0]:
      del tokenGame[0]
    return tokenGame

  def compressGame(self, game):
    str = ""
    for pairMoves in game:
      aMoves = re.split("\\s+", pairMoves) # split into each opponent's moves
      if len(aMoves)==0 or len(aMoves)>2:
        raise Exception("invalid input")

      for i in range(len(aMoves)): # for each opponent
        moves = re.split("[-x]", aMoves[i])
        # TODO: verify that moves are legal
        for j in range(len(moves)): # for each move
          position = int(moves[j])
          if not position:
            raise Exception("invalid input")
          if position<=0 or position>32:
            raise Exception("invalid input")
          ch = self.EnglishNotationMap[position-1] # use our mapping
          str += ch
        str += ','
    return str

  def printBoard(self):
    for i in range(8):
      print(''.join(self.board[i]))
    print()

  # convert into standard notation and print game
  # text - our compressed representation
  def displayGame(self, text2):
    self.printBoard()

    moveNo = 1
    player = '1'
    moves = text2.split(",")
    for move in moves:
      if not move:
        continue
      print("{}) ".format(moveNo), end='')
      startRow = -1
      startCol = -1
      endRow = -1
      endCol = -1
      for ch in list(move):
        position = self.reverseMap(ch)
        if startRow < 0:
          startRow = self.getRow(position)
          startCol = self.getCol(position)
        else:
          endRow = self.getRow(position)
          endCol = self.getCol(position)
        if endRow > 0:
          if abs(endRow-startRow)==1 and abs(endCol-startCol)==1: # regular move
            print("-", end='')
          else:                                                    # jump
            print("x", end='')
            # remove a piece from the board
            row = endRow - 1
            if endRow < startRow:
              row = endRow + 1
            col = endCol - 1
            if endCol < startCol:
              col = endCol + 1
            self.board[row][col] = '*'
        print(position, end='')
      self.board[startRow][startCol] = '*'
      self.board[endRow][endCol] = player
      print()
      self.printBoard();
      if player == '1': # alternate
        player = '2'
      else:
        player = '1'
      moveNo += 1
      if moveNo == 91:
        pass
    pass

####################

def main():
  checkers = Checkers()
  game = checkers.readInput()
  text = checkers.compressGame(game)
  print("compressed game: " + text) # this is what we "save to disk"
  checkers.displayGame(text)

if __name__ == '__main__':
  main()
  print("end.")
  sys.exit(0)
```

Output:

```
  initial game: 1. 9-14 23-18 2. 14x23 27x18 3. 5-9 26-23 4. 12-16 30-26 5. 16-19 24x15 6.
10x19 23x16 7. 11x20 22-17 8. 7-11 18-15 9. 11x18 28-24 10. 20x27 32x5 11. 8-11 26-23 12. 4-8
25-22 13. 11-15 17-13 14. 8-11 21-17 15. 11-16 23-18 16. 15-19 17-14 17. 19-24 14-10 18. 6x15
18x11 19. 24-28 22-17 20. 28-32 17-14 21. 32-28 31-27 22. 16-19 27-24 23. 19-23 24-20 24. 23-26
29-25 25. 26-30 25-21 26. 30-26 14-9 27. 26-23 20-16 28. 23-18 16-12 29. 18-14 11-8 30. 28-24
8-4 31. 24-19 4-8 32. 19-16 9-6 33. 1x10 5-1 34. 10-15 1-6 35. 2x9 13x6 36. 16-11 8-4 37. 15-18
6-1 38. 18-22 1-6 39. 22-26 6-1 40. 26-30 1-6 41. 30-26 6-1 42. 26-22 1-6 43. 22-18 6-1 44.
14-9 1-5 45. 9-6 21-17 46. 18-22
compressed game:
IN,WR,NW,aR,EI,ZW,LP,dZ,PS,XO,JS,WP,KT,VQ,GK,RO,KR,bX,Ta,fE,HK,ZW,DH,YV,KO,QM,HK,UQ,KP,WR,OS,QN
,SX,NJ,FO,RK,Xb,VQ,bf,QN,fb,ea,PS,aX,SW,XT,WZ,cY,Zd,YU,dZ,NI,ZW,TP,WR,PL,RN,KH,bX,HD,XS,DH,SP,I
F,AJ,EA,JO,AF,BI,MF,PK,HD,OR,FA,RV,AF,VZ,FA,Zd,AF,dZ,FA,ZV,AF,VR,FA,NI,AE,IF,UQ,RV,
.1.1.1.1
1.1.1.1.
.1.1.1.1
*.*.*.*.
.*.*.*.*
2.2.2.2.
.2.2.2.2
2.2.2.2.

1) 9-14
.1.1.1.1
1.1.1.1.
.*.1.1.1
*.1.*.*.
.*.*.*.*
2.2.2.2.
```

```
.2.2.2.2
2.2.2.2.

2) 23-18
.1.1.1.1
1.1.1.1.
.*.1.1.1
*.1.*.*.
.*.2.*.*
2.2.*.2.
.2.2.2.2
2.2.2.2.

3) 14x23
.1.1.1.1
1.1.1.1.
.*.1.1.1
*.*.*.*.
.*.*.*.*
2.2.1.2.
.2.2.2.2
2.2.2.2.

4) 27x18
.1.1.1.1
1.1.1.1.
.*.1.1.1
*.*.*.*.
.*.2.*.*
2.2.*.2.
.2.2.*.2
2.2.2.2.

5) 5-9
.1.1.1.1
*.1.1.1.
.1.1.1.1
*.*.*.*.
.*.2.*.*
2.2.*.2.
.2.2.*.2
2.2.2.2.

6) 26-23
.1.1.1.1
*.1.1.1.
.1.1.1.1
*.*.*.*.
.*.2.*.*
2.2.2.2.
.2.*.*.2
2.2.2.2.

7) 12-16
.1.1.1.1
*.1.1.1.
.1.1.1.*
```

```
*.*.*.1.
.*.2.*.*
2.2.2.2.
.2.*.*.2
2.2.2.2.

8) 30-26
.1.1.1.1
*.1.1.1.
.1.1.1.*
*.*.*.1.
.*.2.*.*
2.2.2.2.
.2.2.*.2
2.*.2.2.

9) 16-19
.1.1.1.1
*.1.1.1.
.1.1.1.*
*.*.*.*.
.*.2.1.*
2.2.2.2.
.2.2.*.2
2.*.2.2.

10) 24x15
.1.1.1.1
*.1.1.1.
.1.1.1.*
*.*.2.*.
.*.2.*.*
2.2.2.*.
.2.2.*.2
2.*.2.2.

11) 10x19
.1.1.1.1
*.1.1.1.
.1.*.1.*
*.*.*.*.
.*.2.1.*
2.2.2.*.
.2.2.*.2
2.*.2.2.

12) 23x16
.1.1.1.1
*.1.1.1.
.1.*.1.*
*.*.*.2.
.*.2.*.*
2.2.*.*.
.2.2.*.2
2.*.2.2.

13) 11x20
```

```
.1.1.1.1
*.1.1.1.
.1.*.*.*
*.*.*.*.
.*.2.*.1
2.2.*.*.
.2.2.*.2
2.*.2.2.

14) 22-17
.1.1.1.1
*.1.1.1.
.1.*.*.*
*.*.*.*.
.2.2.*.1
2.*.*.*.
.2.2.*.2
2.*.2.2.

15) 7-11
.1.1.1.1
*.1.*.1.
.1.*.1.*
*.*.*.*.
.2.2.*.1
2.*.*.*.
.2.2.*.2
2.*.2.2.

16) 18-15
.1.1.1.1
*.1.*.1.
.1.*.1.*
*.*.2.*.
.2.*.*.1
2.*.*.*.
.2.2.*.2
2.*.2.2.

17) 11x18
.1.1.1.1
*.1.*.1.
.1.*.*.*
*.*.*.*.
.2.1.*.1
2.*.*.*.
.2.2.*.2
2.*.2.2.

18) 28-24
.1.1.1.1
*.1.*.1.
.1.*.*.*
*.*.*.*.
.2.1.*.1
2.*.*.2.
.2.2.*.*
```

```
2.*.2.2.

19) 20x27
.1.1.1.1
*.1.*.1.
.1.*.*.*
*.*.*.*.
.2.1.*.*
2.*.*.*.
.2.2.1.*
2.*.2.2.

20) 32x5
.1.1.1.1
2.1.*.1.
.*.*.*.*
*.*.*.*.
.2.1.*.*
2.*.*.*.
.2.2.1.*
2.*.2.*.

21) 8-11
.1.1.1.1
2.1.*.*.
.*.*.1.*
*.*.*.*.
.2.1.*.*
2.*.*.*.
.2.2.1.*
2.*.2.*.

22) 26-23
.1.1.1.1
2.1.*.*.
.*.*.1.*
*.*.*.*.
.2.1.*.*
2.*.2.*.
.2.*.1.*
2.*.2.*.

23) 4-8
.1.1.1.*
2.1.*.1.
.*.*.1.*
*.*.*.*.
.2.1.*.*
2.*.2.*.
.2.*.1.*
2.*.2.*.

24) 25-22
.1.1.1.*
2.1.*.1.
.*.*.1.*
*.*.*.*.
```

```
.2.1.*.*
2.2.2.*.
.*.*.1.*
2.*.2.*.

25) 11-15
.1.1.1.*
2.1.*.1.
.*.*.*.*
*.*.1.*.
.2.1.*.*
2.2.2.*.
.*.*.1.*
2.*.2.*.

26) 17-13
.1.1.1.*
2.1.*.1.
.*.*.*.*
2.*.1.*.
.*.1.*.*
2.2.2.*.
.*.*.1.*
2.*.2.*.

27) 8-11
.1.1.1.*
2.1.*.*.
.*.*.1.*
2.*.1.*.
.*.1.*.*
2.2.2.*.
.*.*.1.*
2.*.2.*.

28) 21-17
.1.1.1.*
2.1.*.*.
.*.*.1.*
2.*.1.*.
.2.1.*.*
*.2.2.*.
.*.*.1.*
2.*.2.*.

29) 11-16
.1.1.1.*
2.1.*.*.
.*.*.*.*
2.*.1.1.
.2.1.*.*
*.2.2.*.
.*.*.1.*
2.*.2.*.

30) 23-18
.1.1.1.*
```

```
2.1.*.*.
.*.*.*.*
2.*.1.1.
.2.2.*.*
*.2.*.*.
.*.*.1.*
2.*.2.*.

31) 15-19
.1.1.1.*
2.1.*.*.
.*.*.*.*
2.*.*.1.
.2.2.1.*
*.2.*.*.
.*.*.1.*
2.*.2.*.

32) 17-14
.1.1.1.*
2.1.*.*.
.*.*.*.*
2.2.*.1.
.*.2.1.*
*.2.*.*.
.*.*.1.*
2.*.2.*.

33) 19-24
.1.1.1.*
2.1.*.*.
.*.*.*.*
2.2.*.1.
.*.2.*.*
*.2.*.1.
.*.*.1.*
2.*.2.*.

34) 14-10
.1.1.1.*
2.1.*.*.
.*.2.*.*
2.*.*.1.
.*.2.*.*
*.2.*.1.
.*.*.1.*
2.*.2.*.

35) 6x15
.1.1.1.*
2.*.*.*.
.*.*.*.*
2.*.1.1.
.*.2.*.*
*.2.*.1.
.*.*.1.*
2.*.2.*.
```

```
36) 18x11
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.*.*.1.
.*.*.*.*
*.2.*.1.
.*.*.1.*
2.*.2.*.

37) 24-28
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.*.*.1.
.*.*.*.*
*.2.*.*.
.*.*.1.1
2.*.2.*.

38) 22-17
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.*.*.1.
.2.*.*.*
*.*.*.*.
.*.*.1.1
2.*.2.*.

39) 28-32
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.*.*.1.
.2.*.*.*
*.*.*.*.
.*.*.1.*
2.*.2.1.

40) 17-14
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.1.
.*.*.*.*
*.*.*.*.
.*.*.1.*
2.*.2.1.

41) 32-28
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.1.
.*.*.*.*
```

```
*.*.*.*.
.*.*.1.1
2.*.2.*.

42) 31-27
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.1.
.*.*.*.*
*.*.*.*.
.*.*.2.1
2.*.*.*.

43) 16-19
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.*.
.*.*.1.*
*.*.*.*.
.*.*.2.1
2.*.*.*.

44) 27-24
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.*.
.*.*.1.*
*.*.*.2.
.*.*.*.1
2.*.*.*.

45) 19-23
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.*.
.*.*.*.*
*.*.1.2.
.*.*.*.1
2.*.*.*.

46) 24-20
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.*.
.*.*.*.2
*.*.1.*.
.*.*.*.1
2.*.*.*.

47) 23-26
.1.1.1.*
2.*.*.*.
```

```
.*.*.2.*
2.2.*.*.
.*.*.*.2
*.*.*.*.
.*.1.*.1
2.*.*.*.

48) 29-25
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.*.
.*.*.*.2
*.*.*.*.
.2.1.*.1
*.*.*.*.

49) 26-30
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.*.
.*.*.*.2
*.*.*.*.
.2.*.*.1
*.1.*.*.

50) 25-21
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.*.
.*.*.*.2
2.*.*.*.
.*.*.*.1
*.1.*.*.

51) 30-26
.1.1.1.*
2.*.*.*.
.*.*.2.*
2.2.*.*.
.*.*.*.2
2.*.*.*.
.*.1.*.1
*.*.*.*.

52) 14-9
.1.1.1.*
2.*.*.*.
.2.*.2.*
2.*.*.*.
.*.*.*.2
2.*.*.*.
.*.1.*.1
*.*.*.*.
```

```
53) 26-23
.1.1.1.*
2.*.*.*.
.2.*.2.*
2.*.*.*.
.*.*.*.2
2.*.1.*.
.*.*.*.1
*.*.*.*.

54) 20-16
.1.1.1.*
2.*.*.*.
.2.*.2.*
2.*.*.2.
.*.*.*.*
2.*.1.*.
.*.*.*.1
*.*.*.*.

55) 23-18
.1.1.1.*
2.*.*.*.
.2.*.2.*
2.*.*.2.
.*.1.*.*
2.*.*.*.
.*.*.*.1
*.*.*.*.

56) 16-12
.1.1.1.*
2.*.*.*.
.2.*.2.2
2.*.*.*.
.*.1.*.*
2.*.*.*.
.*.*.*.1
*.*.*.*.

57) 18-14
.1.1.1.*
2.*.*.*.
.2.*.2.2
2.1.*.*.
.*.*.*.*
2.*.*.*.
.*.*.*.1
*.*.*.*.

58) 11-8
.1.1.1.*
2.*.*.2.
.2.*.*.2
2.1.*.*.
.*.*.*.*
2.*.*.*.
```

```
.*.*.*.1
*.*.*.*.

59) 28-24
.1.1.1.*
2.*.*.2.
.2.*.*.2
2.1.*.*.
.*.*.*.*
2.*.*.1.
.*.*.*.*
*.*.*.*.

60) 84
.1.1.1.2
2.*.*.*.
.2.*.*.2
2.1.*.*.
.*.*.*.*
2.*.*.1.
.*.*.*.*
*.*.*.*.

61) 24-19
.1.1.1.2
2.*.*.*.
.2.*.*.2
2.1.*.*.
.*.*.1.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

62) 4-8
.1.1.1.*
2.*.*.2.
.2.*.*.2
2.1.*.*.
.*.*.1.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

63) 19-16
.1.1.1.*
2.*.*.2.
.2.*.*.2
2.1.*.1.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

64) 9-6
.1.1.1.*
2.2.*.2.
.*.*.*.2
```

```
2.1.*.1.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

65) 1x10
.*.1.1.*
2.*.*.2.
.*.1.*.2
2.1.*.1.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

66) 51
.2.1.1.*
*.*.*.2.
.*.1.*.2
2.1.*.1.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

67) 10-15
.2.1.1.*
*.*.*.2.
.*.*.*.2
2.1.1.1.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

68) 1-6
.*.1.1.*
*.2.*.2.
.*.*.*.2
2.1.1.1.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

69) 2x9
.*.*.1.*
*.*.*.2.
.1.*.*.2
2.1.1.1.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

70) 13x6
```

```
.*.*.1.*
*.2.*.2.
.*.*.*.2
*.1.1.1.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

71) 16-11
.*.*.1.*
*.2.*.2.
.*.*.1.2
*.1.1.*.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

72) 84
.*.*.1.2
*.2.*.*.
.*.*.1.2
*.1.1.*.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

73) 15-18
.*.*.1.2
*.2.*.*.
.*.*.1.2
*.1.*.*.
.*.1.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

74) 61
.2.*.1.2
*.*.*.*.
.*.*.1.2
*.1.*.*.
.*.1.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

75) 18-22
.2.*.1.2
*.*.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.1.*.*.
.*.*.*.*
```

```
*.*.*.*.

76) 1-6
.*.*.1.2
*.2.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.1.*.*.
.*.*.*.*
*.*.*.*.

77) 22-26
.*.*.1.2
*.2.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.*.*.*.
.*.1.*.*
*.*.*.*.

78) 61
.2.*.1.2
*.*.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.*.*.*.
.*.1.*.*
*.*.*.*.

79) 26-30
.2.*.1.2
*.*.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.1.*.*.

80) 1-6
.*.*.1.2
*.2.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.*.*.*.
.*.*.*.*
*.1.*.*.

81) 30-26
.*.*.1.2
*.2.*.*.
.*.*.1.2
*.1.*.*.
```

```
.*.*.*.*
2.*.*.*.
.*.1.*.*
*.*.*.*.

82) 61
.2.*.1.2
*.*.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.*.*.*.
.*.1.*.*
*.*.*.*.

83) 26-22
.2.*.1.2
*.*.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.1.*.*.
.*.*.*.*
*.*.*.*.

84) 1-6
.*.*.1.2
*.2.*.*.
.*.*.1.2
*.1.*.*.
.*.*.*.*
2.1.*.*.
.*.*.*.*
*.*.*.*.

85) 22-18
.*.*.1.2
*.2.*.*.
.*.*.1.2
*.1.*.*.
.*.1.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

86) 61
.2.*.1.2
*.*.*.*.
.*.*.1.2
*.1.*.*.
.*.1.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

87) 14-9
.2.*.1.2
```

```
*.*.*.*.
.1.*.1.2
*.*.*.*.
.*.1.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

88) 1-5
.*.*.1.2
2.*.*.*.
.1.*.1.2
*.*.*.*.
.*.1.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

89) 9-6
.*.*.1.2
2.1.*.*.
.*.*.1.2
*.*.*.*.
.*.1.*.*
2.*.*.*.
.*.*.*.*
*.*.*.*.

90) 21-17
.*.*.1.2
2.1.*.*.
.*.*.1.2
*.*.*.*.
.2.1.*.*
*.*.*.*.
.*.*.*.*
*.*.*.*.

91) 18-22
.*.*.1.2
2.1.*.*.
.*.*.1.2
*.*.*.*.
.2.*.*.*
*.1.*.*.
.*.*.*.*
*.*.*.*.

end.
```