Iverson computing competition 2017 monday may 29 with solutions

name	______________________________
school	______________________________
city	_______________
grade	_______
cs teacher	______________________________

illegible answers will not be marked

question	– – – –	marks	your score
1	divisors	10	
2	solitaire	10	
3	counterfeit	11	
4	teams	10	
total	– – – –	41	

Exam and solutions prepared by Zachary Friggstad and Ryan Hayward

Exam Format

This is a two-hour paper and pencil exam. There are four questions, each with multiple parts. Solve as many parts of as many questions as you can.

Programming Language

Questions that require programming can be answered using any language (e.g. C/C++, Java, Python, ...) or pseudo-code. **Pseudo-code should be detailed enough to allow for a near direct translation into a programming language.** Clarify your code with appropriate comments. For full marks, an answer must be correct, well-explained, and as simple as possible.

Our primary interest is in thinking skill rather than coding wizardry, so logical thinking and systematic problem solving count for more than programming language knowledge.

Suggestions

1. You can assume that the user enters only valid input in the coding questions.
2. In somes cases, sample executions of the desired program are shown. Review the samples carefully to make sure you understand the specifications. The samples may give hints.
3. Design your algorithm before writing any code. Use any format (pseudo-code, diagrams, tables) or aid to assist your design plan. We may give part marks for legible rough work, especially if your final answer is lacking. We are looking for key computing ideas, not specific coding details, so you can invent your own "built-in" functions for simple subtasks such as reading the next number, or the next character in a string, or loading an array. Make sure to specify such functions by giving a relationship between their inputs and outputs.
4. Read all questions before deciding which ones to attempt, and in which order. Start with the easiest parts of each question.

question 1: divisors

An *integer* is a whole number, e.g. -17 or 0 or 3926. For integers d, n, say d is a *divisor* of n if there is an integer k such that $n = k \times d$. E.g., 7 is a divisor of 21 because $21 = 3 \times 7$, and -917 is a divisor of 0 because $0 = 0 \times -917$. For integers x, y, the *greatest common divisor* of x and y, written gcd(x, y), is the largest integer d that is a divisor of both x and y. E.g., gcd(15,12) = 3, gcd$(-7, -5)$ = 1, and gcd(4,0) = 4.

a) [1 mark] List all divisors of 15.

b) [1 mark] List all divisors of -12. Hint: gcd(15,-12) is 3.

c) [1 mark] The value gcd(0,0) is not defined because **(circle the best answer)**

i) every integer is a common divisor of these two integers
ii) division by zero is not defined
iii) the limit of the minimum integer that divides both tends to minus infinity
iv) any algorithm to compute this gcd will not terminate

```
def gcd(x,y):
  assert(x>0 and y>=0 and x>=y)
  ops = 0                    # count division/remainder ops
  if y==0:
    return x, ops
  divisors = []              # empty list
  for d in range(1,y+1):     # d ranges from 1 to y
    ops += 1                 # ops = ops + 1
    if (0 == (x%d)) and (0 == (y%d)):  # x%d is remainder of x divided by d
      divisors.append(d)     # append d to list
  return max(divisors), ops
```

In this code, `x%d` returns the remainder after integer division `x//d`. E.g. `15%6` returns 3, since $15 = 2 \times 6 + 3$. The above function returns gcd(x,y) and the number of remainder calls. Output from `print(15, 12, gcd(15,12) )` is `15 12 (3, 12)` .

d) [1 mark] Give the output for `print(30, 7, gcd(30,7) )`.

```
def foo(x,y):
  assert(x>0 and y>=0 and x>=y)
  ops = 0
  while y > 0:
    ops += 1
    x, y = y, x % y  # simultaneous assignment x=y, y= x%y
  return x, ops
```

```
output from print(foo(30,7)):          output from print(foo(44,28)):
30
7 2
2 1
1 0
(1, 3)
```

e) [1 mark] foo(x,y) returns the same values as gcd(x,y), but prints more output, shown above. Above, give the output from `print(foo(44,28))`.

f) [2 marks] To show that foo returns gcd(x,y), a useful property is this: for integers d, x, y, if d divides x and y then d divides $x - y$. Prove this property.

g) [3 marks] **Circle the best answer** and explain briefly why your answer is correct. Hint: for $0 < y < x$, `(x%y)` $< x/2$. Also, $2^{20} = 1048576$.

i) The last line of output from `print(foo(832040,514229))` is (1, 8).
ii) The last line of output from `print(foo(832040,514229))` is (1, 28).
iii) The last line of output from `print(foo(832040,514229))` is (1, 48).
iv) The last line of output from `print(foo(832040,514229))` is (1, 68).

question 2: solitaire

Solitaire is a 1-player stone-jumping game. A *move* (x,y) consists of jumping a stone over exactly one neighbouring stone into an empty space; the jumped-over stone is removed. (x,y) represents the jump from location x to y. We use non-negative integers for locations, so for move (x,y) x−y is +2 or −2 and the stone at location $(x + y)//2$ is removed. E.g. for the left position below, possible moves are (2, 0), (1, 3), (6, 4).

```
a position           position after move (1, 3)      target position
0 1 2 3 4 5 6             0 1 2 3 4 5 6               0 1 2 3 4 5 6
- x x - - x x             - - x x - x x               - - - x - - -
```

The goal of solitaire is to find a a move sequence from start position to target. Such a sequence is a *solution*. The target can be any position. E.g. start `x-xx-` and target `--x--` has solution (3,1), (0,2). E.g. start `-xx--` and target `--x--` has no solution.

a) [2 marks] For start `xx---xx` and target `---x---`, find a solution or explain why there is none.

b) [2 marks] Repeat a) for start `xxxx--xx`, target `----x---`.

```
def legal_moves(brd): # s is stone
  moves = []
  for j in range(len(brd)-2):
    if brd[j] == s and brd[j+1] == s and brd[j+2] != s: moves.append((j,j+2))
    if brd[j] != s and brd[j+1] == s and brd[j+2] == s: moves.append((j+2,j))
  return moves
```

c) [1 mark] The above code generates legal moves. E.g. `legal_moves('-xx-xx')` outputs `[(2, 0), (1, 3), (5, 3)]`. Give output from `legal_moves('-x-xx-xx-')`.

```
def solve(brd, target, indent):
  print(indent*'. ' + brd)
  if brd==target: return True, []
  moves = legal_moves(brd)
  if 0==len(moves): return False, []
  for m in moves:
    b_new = make_move(brd,m)
    result, sequence = solve(b_new,target, indent+1)
    if result:
      sequence.insert(0,(m)) # add m to front of sequence
      return True, sequence
  return False, []
```

```
solve('-x-xx', '--x--', 0)      solve('-xx-x', '--x--', 0)
-x-xx                           -xx-x
. -xx--                         . x---x
. . x----                       . ---xx
. . ---x-                       . . --x--
(False, [])                     (True, [(1, 3), (4, 2)])
```

d) [2 marks] Above is solving code, with two sample outputs.
Show output from `solve('xx-xx', '--x--', 0)`.

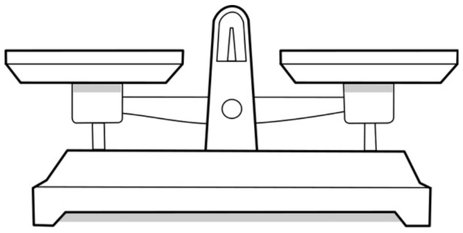
e) [3 marks] Below, show the rest of the output.

```
for rough work

solve('xx-xxx-x', '---x----', 0)

xx-xxx-x

. --xxxx-x
```

```
your final answer

solve('xx-xxx-x', '---x----', 0)

xx-xxx-x

. --xxxx-x
```

question 3: counterfeit

You have coins labelled 0 through $n-1$. They look the same, but **exactly one** is counterfeit: it has a slightly different weight than that of a genuine coin. All genuine coins weigh the same. You will find the counterfeit using a balance scale.



A *weighing* consists of putting some number of coins on one side of the scale and the same number of coins on the other side. The scale then reports either `balanced` or `unbalanced`. When unbalanced, the scale does not report which side is lighter. Your goal is to find the counterfeit using the fewest weighings.

Example: after the two weighings below, we know that 0,3 are genuine and either 1 or 4 (and so not 2) is counterfeit. So after these weighings we call 1,4 *possible counterfeit*.

`left side`	`right side`	`outcome`
`0`	`3`	`balanced`
`0 1`	`3 4`	`unbalanced`

(a) [1 marks] After this weighing sequence, list all possible counterfeits among {0,1,2,3,4}.

`left side`	`right side`	`outcome`
`0 1`	`2 3`	`unbalanced`
`0 2`	`1 4`	`unbalanced`
`4`	`2`	`balanced`

Possible counterfeit(s): ____________________
Explain briefly.

(b) [2 marks] After this weighing sequence, list all possible counterfeits among {0,1,2,. . . ,9}.

left side	right side	outcome
9 2 7 1	3 8 6 5	unbalanced
1 5 4 3	7 2 0 8	unbalanced
7 0	6 5	balanced
1 2 3 8	4 5 6 7	unbalanced
3 7	2 9	balanced
9 6 8 2	3 5 7 0	unbalanced

Possible counterfeit(s): ____________________
Explain briefly.

(c) [1 marks] For 4 coins, explain how to find the counterfeit with at most 2 weighings.

(d) [1 marks] For 6 coins, explain how to find the counterfeit with at most 3 weighings.

(e) [3 marks] Recall that there is exactly one counterfeit. Complete the pseudocode below: given a sequence of weighings among n coins, print the label ℓ of each possible counterfeit (i.e. each ℓ such that, if ℓ is counterfeit, then the result of each weighing is exactly as in `res`).

```
# input:
# n - number of coins, m - number of measurements
# lhs, rhs - lists of m measurements
# res - list of m outcomes, "b" for balanced, "u" for unbalanced
#
# output: print the possible counterfeits, in any order
#
# For 0 <= i < m, lhs[i], rhs[i] are lists of coins used in the i'th measurement.
# You may assume lhs[i] and rhs[i] have the same length, only include
# coins from 0 to n-1, and no coin appears in both lhs[i] and rhs[i].
#
# Use len(lhs[i]) to denote the number of coins in lhs[i], etc.
#
# example:
# find_counterfeit(5, 2, [[0], [0,1]] , [[3], [3, 4]], ["b", "u"])
# This corresponds to the first example above.
# output: 1 4

find_counterfeits(n, m, lhs, rhs, res)
```

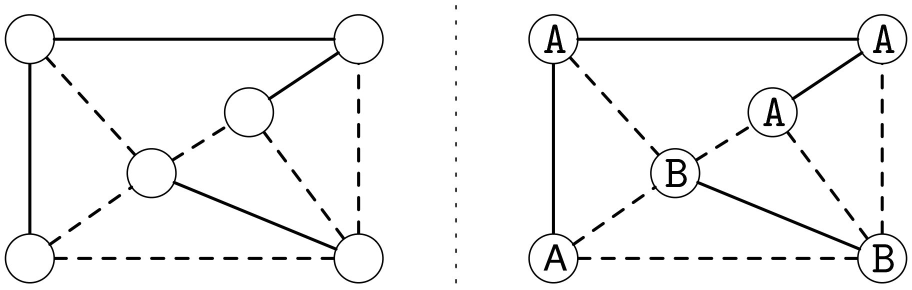
(f) [2 or 3 marks]: For any integer $k \geq 2$, for $n = 2^k$ coins, describe how to find the counterfeit with at most k weighings. A correct strategy is worth 2 marks if it is adaptive (i.e. the weighing sequence will vary depending on the results of the weighings), or 3 marks if it is non-adaptive (i.e. the weighing sequence does not depend on the results). Hint: binary.

question 4: teams

You want to form two soccer teams. Some player pairs prefer to be mates (i.e. teammates), some pairs prefer to be opponents, and some pairs have no preference. Given n players numbered 0 through $n-1$ and a list of mate/opponent preferences, you should assign each player to team `A` or `B` so that all preferences are satisfied, if possible. (The teams do not have to have the same number of players.)

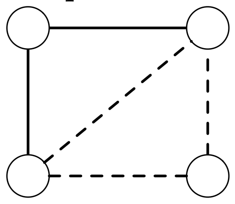
An input instance can be drawn as a **graph**: each player is a node (circle), each mate preference is a solid line, each opponent preference is a dashed line.

Example: The left graph shows players and preferences, the right graph also shows an assignment of players to teams.

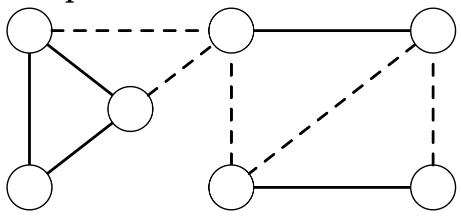


(a) [3 marks] In each following example, label nodes `A` or `B` (exactly one label each) so preferences are satisfied. If this is impossible, explain using the space beside the graph (if it helps with your explanation, you may also draw on the graph).

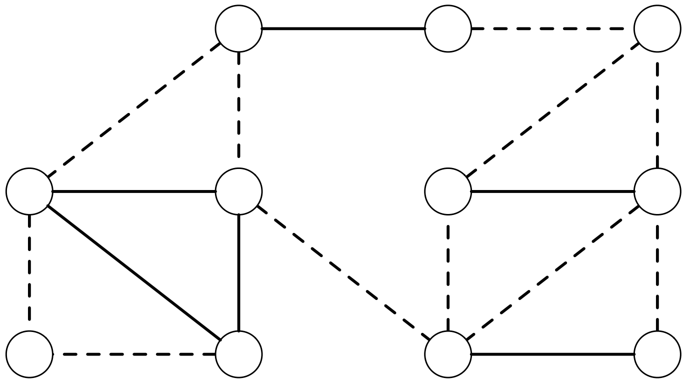
Graph 1



Graph 2

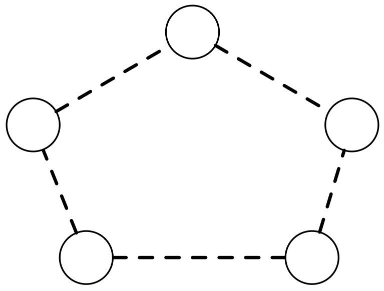


Graph 3



(b) [2 marks] Suppose all edges are dashed, i.e. all preferences are opponents. An *opponent cycle* is a sequence of players $k_1, k_2, \dots, k_c$ where k_i and k_{i+1} are preferred opponents for each $1 \leq i < c$ and k_1 and k_c are also preferred opponents. Say c is the length of the opponent cycle, so the picture has length 5.

Picture of a cycle:



Describe which opponent cycles of length $c \geq 3$ can be partitioned into 2 teams satisfying all preferences on the cycle. Be clear and concise.

(c) [3 marks] On the next page, describe in pseudocode how to determine if a given graph of preferences can have its players partitioned into 2 teams so all mate and opponent preferences are satisfied. Make these assumptions: players are numbered 0 through $n-1$; there is a given 2-dimensional array `graph` indexed from 0 to $n-1$ where `graph[i][j]` is 1 if i and j are preferred mates, -1 if i and j are preferred opponents, and 0 if they are have no preference; always, `graph[i][i] = 0` for any player i and `graph[i][j] = graph[j][i]` for any pair of players i, j.

The output for your program is just `possible` or `impossible`, depending on whether two teams can be formed to satisfy all preferences.

```
# example, n = 4 and             corresponds to graph in part a)
# graph = [
#   [  0,  1,  1,  0],
#   [  1,  0, -1, -1],
#   [  1, -1,  0, -1],
#   [  0, -1, -1,  0]
# ]

can_make_teams(n, graph)
```

(d) [2 marks] It is not always possible to satisfy all relationships, but we still want to form two teams. Describe why it is always possible to satisfy at least half of the preferences. Be clear and concise.

solutions

1a) 1,3,5,15,−1,−3,−5,−15

1b) 1,2,3,4,6,12,−1,−2,−3,−4,−6,−12

1c) i)

1d) 30, 7, (1, 7)

```
1e)   30 7 (1, 7)
      44
      28 16
      16 12
      12 4
      4 0
      (4, 4)
```

1f) Assume d divides x and d divides y. Then by definition there exists an integer, say t, such that $x = dt$; also, there exists an integers, say w, such that $y = dw$. So $x - y = dt - dw = d(t - w)$. Since t and w are integers, $t - w$ is an integer. So there exists an integer r, where $r = t - w$, such that $x - y = dr$. So, by definition, d divides $x - y$.

1g) Every two iterations of `foo`, `x` is replaced with `x%y`. So the number of iterations is at most $2\times$ $\log_2$ of `x`, so at most 40. In the first 8 iterations, x has value 514229 317811 196418 121393 75025 46368 28657 17711, so the number of iterations is more than 8. So the correct answer is ii).

2a) The only move sequences from `xx---xx` are `--x--xx` and then `--x-x--` or `xx--x--` and then `--x-x--`, so there is no solution.

2b) One sequence is (2,4), (7,5), (0,2), (5,3), (2,4). There are other correct answers, e.g. the first two moves can be exchanged, and also the next two.

2c) `[(4,2), (3,5), (7,5), (6,8)]` is the only correct answer.

```
2d) xx-xx
    . --xxx
    . . -x--x
    . xxx--
    . . x--x-
    (False, [])
```

```
2e) xx-xxx-x
    . --xxxx-x
    . . -x--xx-x
    . . . -x-x---x
    . . . -x----xx
    . . . . -x---x--
    . . --xx--xx
    . . . -x----xx
    . . . . -x---x--
    . . . ----x-xx
    . . . . ----xx--
    . . . . . ---x----
    (True, [(0, 2), (4, 6), (2, 4), (7, 5), (5, 3)])
```

3a) 0 and 1

Coins 2 and 4 are balanced (3rd weighing) and 3 is not unbalanced (2nd weighing). However, 0 or 1 being the counterfeit coin would produce the same outcomes from these weighings.

3b) 8

The balanced weighings rule out coins 0, 2, 3, 5, 7, and 9 leaving only coins 1, 4, or 8 as possible counterfeits. The last weighing rules out coin 1 and the first weighing rules out coin 4. So 8 is the counterfeit (and you can check all weighings are consistent with 8 being counterfeit).

3c) Weighing coins 1 and 2 will determine if the counterfeit is one of 1, 2 or one of 3, 4. In either case, weighing coins 1 and 3 will also tell if the counterfeit is one of 1, 3 or 2, 4. Note that for every pair of coins that was determined to contain an unbalanced coin, precisely one coin lies in both sets. So we uniquely identify the coin.

3d) Weigh coins 1, 2 against coins 3, 4. If this is unbalanced, apply the strategy from part (c) to these coins. Otherwise, we know either 5 or 6 is counterfeit. So weighing coins 1 and 5 will identify the counterfeit.

3e) One approach is to try each coin and see if it being counterfeit would produce the given weighings. Here is working `python3` code that solves the problem.

```
def find_counterfeits(n, m, lhs, rhs, res):
    fake = set()
    for coin in range(n):
        maybe_fake = True
        for i in range(m):
            if res[i] == "b" and coin in lhs[i]+rhs[i]:
                maybe_fake = False
            if res[i] == "u" and coin not in lhs[i]+rhs[i]:
                maybe_fake = False
        if maybe_fake:
            fake.add(coin)
    print(*fake)
```

3f) If $k = 2$ then employ the 4-coin strategy from part (c). Otherwise, if $k \geq 3$ pick any 2^{k-1} coins and weigh them in two equal-size groups (this is possible because $k \geq 3$). If this is unbalanced, discard the other 2^{k-1} coins not weighed. If this is balanced, discard these 2^{k-1} coins.

In either case, we have 2^{k-1} coins remaining. Iterate this strategy until 4 coins remain. Each iteration, the number of coins is halved so after $k - 2$ iterations we are left with 4 coins, and 2 more weighings will find the counterfeit.

For the full 3 marks, the coins in each weighing had to be determined before actually performing them.

Here is the trick: number the coins from 0 to $2^k - 1$ and consider the binary expansion of these numbers. There will be one weighing for each "position" in the binary expansion. The weighing for bit position i (for $0 \leq i < k$) consists of all coins with a 1 at position i in its binary expansion.

Example: consider $k = 4$ and coin $13 = 1101_2 = 2^3 + 2^2 + 2^0$. Include coin 13 in weighings 0, 2 and 3.

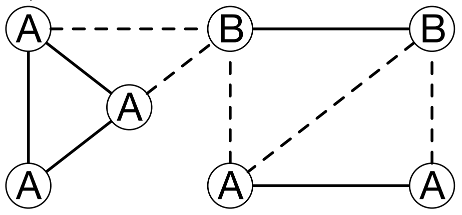
There are exactly 2^{k-1} coins in each weighing and $k \geq 2$ so they can be split into two equal-size groups to determine if the fake coin is involved in that weighing.

The cool part is that if we think of an outcome of *balanced* as 0 and an outcome of *unbalanced* as 1 then forming the binary string of outcomes produces the binary number of the fake coin!

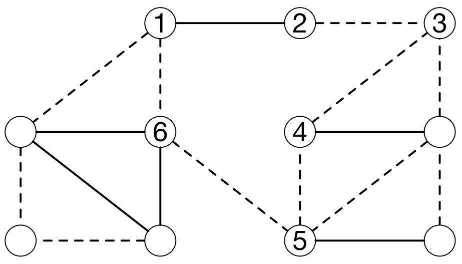
That is, if coin i is the fake then every weighing including i (corresponding to bit positions for i with a 1) will be unbalanced and every weighing not including i (corresponding to bit positions for i with a 0) will be balanced.

4a.i) Impossible, the three players connected by solid edges must be on the same team but this would cause the diagonal opponent edge to not be satisfied.

4a.ii)



4a.iii) Impossible. Consider the sequence of players numbered 1, 2, 3, 4, 5, 6 in the picture below. If 1 is on team A, then 2 is on team A, so 3 is on team B and so on. This forces 6 to be on team A, but 1 and 6 are supposed to be opponents.



4b) It is possible if c is even and impossible if c is odd. If c is even, then just alternate teams around the cycle. If c is odd, we still have to alternate teams around the cycle but then the first and last player will be on the same team.

4c) Here is the idea.

Suppose for the moment that the entire graph is connected (i.e. player 0 can reach any other player following a sequence of edges in the graph). Then it does not matter which team player 0 goes on so we might as well put them on team A. Then iterate the following process: while there is a relationship i, j where i is on a team but j is not then put j on the team that would satisfy the relationship. This will put all players on a team because the graph is connected.

Note this is the only way to assign players to teams given that player 0 is on team A: our choice for each player assignment was forced by the relationship. Now we just have to check that all relationships are satisfied.

If the graph is not connected, then we still do the above. But not all players will be assigned to a team, so repeat the process except choose an unassigned player instead of player 0 to be initially assigned to team A. Repeat until all players are assigned.

Pseudocode

```
while some player k is not on a team
  team[k] = A
  while there is a pair i,j with:
        - g[i][j] != 0
        - team[i] defined
        - team[j] not defined
      then assign team[j] to the team A or B to satisfy the relationship g[i][j]
if all relationships are satisfied
  output "possible"
else
  output "impossible"
```

(continued on next page)

Here is working `python3` code that will correctly solve the problem.

```
def can_make_teams(n, graph):
    team = dict() # team A is 1 and team B is -1

    for k in range(n):
        if k not in team: # i.e. if team[k] is not yet defined
            team[k] = 1
            while True: # repeat until no change
                found = False
                for i in range(n):
                    for j in range(n):
                        if graph[i][j] and i in team and j not in team:
                            found = True # a new team is assigned
                            team[j] = team[i] * graph[i][j]
                if not found:
                    break # exit the loop, no more changes

    ok = True
    for i in range(n):
        for j in range(n):
            if graph[i][j] and team[i] != team[j]*graph[i][j]:
                ok = False
    if ok:
        print("possible")
    else:
        print("impossible")
```

4d) Add players to teams one at at time in any order. When considering a new player, say i, check all preferences it has with players already assigned to a team. Choose the team for i that satisfies at least half of these preferences. Overall this will satisfy half of the preferences.