Computer Science Junior Qualifying Exam

Morning CS1 & CS2 Session

10am-12pm, August 26, 2020

Computer Science Department, Reed College

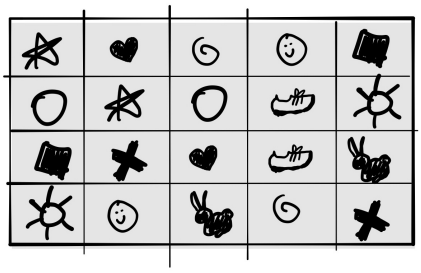
You have two hours to complete this portion of the exam. There are five problems on six pages. Handwrite your answers on blank sheets of paper, writing your name and the number of the problem you are answering clearly on each sheet.

Complete each problem to the best of your ability. When the problem requests that you devise a program or a fragment of a program's code in a certain programming language you should do your best to produce working code in that language — syntactically correct and runnable — that meets the specification of the problem.

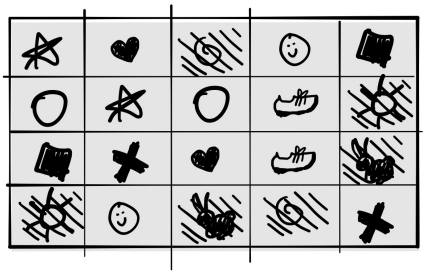
You may ask members of the Reed computer science faculty for clarification on a problem's statement. You cannot, however, use an IDE, a compiler, the web, your notes, etc. to complete the exam, nor can you consult any other individuals for help. Doing so is a violation of the honor principle.

Note that you are welcome to write additional functions that support the code you are asked to write, should you feel the need to do so. (If the instructions ask you to "Write a function that..." that does not mean you cannot write some "helper" functions, as well.)

1. Below is a 4×5 grid of shapes, where each shape has a matching partner in the grid.



Imagine playing a game where you find and match each shape pair. Below we've matched and crossed out three of those pairs.



Let's develop Python code to perform this matching game. Instead of playing the game with shapes, we use integers. We model the game grid as a Python list of grid rows. Each grid row is a Python list of integers. So a 4×5 grid would be a list of 4 grid rows. Each such grid row would be a list of 5 integers. To manage game play, we replace pairs of entries with the Python value `None` to "cross them out." The grid below

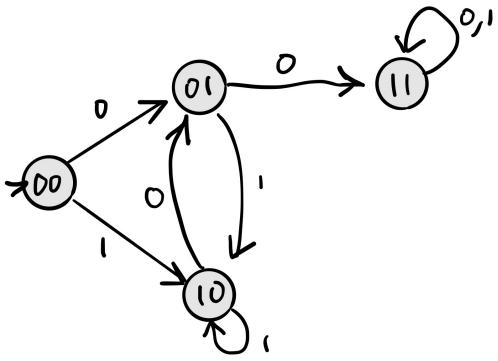
5	3	None	4	2
1	5	1	6	None
2	8	3	6	None
None	4	None	None	8

would be represented by the Python list

```
[[5,3,None,4,2],
 [1,5,1,6,None],
 [2,8,3,6,None],
 [None,4,None,None,8]]
```

Write Python code for a function `def matchNextPair(grid, rows, columns)`. It can assume that `grid` is a list of `rows` lists, and that each row list has length `columns`. Each list entry is an integer or the value `None`. Your function should return `True` if a matching pair of integer entries can be found, and it should replace those two entries with `None`. **Your code can assume each integer entry has a matching entry.** If there are is no matching pair of integers in the grid (all entries are `None`), it should return `False`.

2. Below is a diagram of a finite state machine that has four states named `00`, `01`, `10`, `11`. The transition arrows indicate its behavior when it reads an input bit. For example, if it is in state `01` and it gets fed the bit `1` as input it goes into state `10`.



This table summarizes the behavior given in the diagram:

p_1	p_0	i	q_1	q_0
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

In the above, we use p_1p_0 to represent the state of the machine before it reads the next bit input i, and we use q_1q_0 to represent the state after. So, for example, the 4th line tells us that when the machine is in state `01` and it gets fed the bit `1` as input it goes into state `10`.

Give two boolean expressions using only the logical connectives AND, OR, and NOT to describe this behavior. The first expression should give q_1 as a boolean function of p_1, p_0, and i. The second expression should give q_0 as a boolean function of p_1, p_0, and i.

For full credit, **these two boolean expressions should be** in "sum of products" form.

3. The bottom of this page has a summary of the MIPS32 instruction set, but only for a limited number of its instructions.

   Using only the instructions listed in this guide, write a snippet of MIPS assembly code that computes the integer quotient and the integer remainder due to the division of two integers. Your code should act assuming that two registers are already set to the values involved in the division, just before your code gets executed. Assume that register `$a0` already holds the number to be divided. Assume that register `$a1` holds the divisor. When your code completes execution, register `$v0` should hold the quotient and `$v1` should hold the remainder.

   You can assume that the divisor value in `$a1` is a positive integer.

   For example, if `$a0` holds 38 and `$a1` holds 10, then `$v0` should hold 3 and `$v1` should hold 8 after the code snippet runs. If instead `$a0` holds -38 and `$a1` holds 10, then `$v0` should hold -4 and `$v1` should hold 2 after the code snippet runs. (The latter is because $-4 \times 10 + 2 = -38$. Remainders are non-negative and less than the divisor.)

   For partial credit, get the code working just for the case that `$a1` is non-negative. Then solve the general case, no restriction on `$a1`, for full credit.

   You may only use the instructions given by the guide below.

   MIPS32 coding guide

`li` $RD, *value*	–loads an immediate value into a register.
`add` $RD, $RS1, $RS2,	–add two registers, storing the sum in another.
`sub` $RD, $RS1, $RS2,	–subtract two registers, storing the difference in a third.
`addi` $RD, $RS, *value*	–add a value to a register, storing the sum in another.
`move` $RD, $RS	–copy a register's value to another.
`b` *label*	–jump to a labelled line.
`blt` $RS1, $RS2, *label*	–jump to a labelled line if one register's value is less than another.
`bltz` $RS, *label*	–jump to a labelled line if a register's value is less than zero.
`gt`, `le`, `ge`, `eq`, `ne`	–other conditions than `lt`

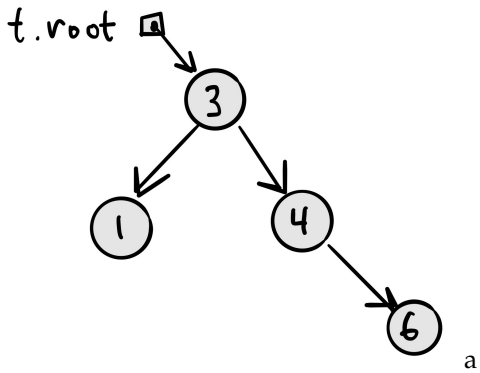
   The registers you can access are named `$v0-v1`, `$a0-a3`, `$t0-t9`, `$s0-s7`, `$sp`, `$fp`, and `$ra`.

4. (a) Below are two C++ definitions of structs that can be used to represent a binary search tree:

```
struct bstNode {
  int key;
  struct bstNode *left;
  struct bstNode *right;
};

struct bst {
  bstNode *root;
};
```

Here is a picture of a four-node binary search tree:



a

The C++ code below allocates four nodes onto the stack frame as `n1`, `n3`, `n4`, and `n6`. It then allocates stack storage for an empty tree `t`:

```
bstNode n1 {1,nullptr,nullptr};
bstNode n3 {3,nullptr,nullptr};
bstNode n4 {4,nullptr,nullptr};
bstNode n6 {6,nullptr,nullptr};
bst t {nullptr};
```

Write additional lines of C++ code to link together the nodes and set the `root` attribute of `t`. The code should make a tree data structure that **mimics the picture**.

(b) **Write a C++ function** `void outputInOrder(bstNode* n)` so that it outputs the keys of the binary search tree rooted at `n` in order from least to greatest key, one per line of `std::cout`. For example, if we continued your C++ code that we wrote above with the line

```
outputInOrder(t.root);
```

It should output the four lines

```
1
3
4
6
```

Note that your code should handle the case when `n` is `nullptr`. In that case it outputs no lines.

5. (a) **Write a Python function** `def makeDictQuery(d,v)` that takes a Python dictionary `d` and a value `v`. That value will be used as a "default value" for when `d` is missing an entry.

   This function **should return a function** that is able to query the dictionary `d` and look up its entries. For the sake of describing its behavior, let's call it `dQuery`. When `dQuery` is given a key that has an entry in `d`, it should return that entry's value. When `dQuery` is given a key for which `d` has no entry, it should return `v`.

   For example, consider the following Python interaction:

```
>>> d1 = {'abe':2, 'bar':4, 'carlos':1}
>>> q1 = makeDictQuery(d1,-1)
>>> q1('abe')
2
>>> q1('dolly')
-1
>>> d2 = {'x':23, 'y':32, 'z':101}
>>> q2 = makeDictQuery(d2,None)
>>> print(q2('z'))
101
>>> print(q2('g'))
None
```

   (b) **Now write a somewhat similar Python function** `def makeQueryMaker(d)` that takes a Python dictionary `d`. It should return a function that, when handed a default value `v`, **returns a function** that is able to query `d` and look up its entries.

   To make this a little clearer, consider the following Python interaction:

```
>>> d1 = {'abe':2, 'bar':4, 'carlos':1}
>>> q1maker = makeQueryMaker(d1)
>>> q1minusOne = q1maker(-1)
>>> q1oneThousand = q1maker(1000)
>>> q1minusOne('abe')
2
>>> q1minusOne('dolly')
-1
>>> q1oneThousand('abe')
2
>>> q1oneThousand('dolly')
1000
```

   In the above interaction, we use `q1maker` twice to introduce two different querying functions for the dictionary `d1`, one with a default value of -1, the other with a default value of 1000.

Computer Science Junior Qualifying Exam
Afternoon CS1 & CS2 Session

1-3pm, August 26, 2020

Computer Science Department, Reed College

You have two hours to complete this portion of the exam. There are five problems on six pages. Handwrite your answers on blank sheets of paper, writing your name and the number of the problem you are answering clearly on each sheet.

Complete each problem to the best of your ability. When the problem requests that you devise a program or a fragment of a program's code in a certain programming language you should do your best to produce working code in that language — syntactically correct and runnable — that meets the specification of the problem.

You may ask members of the Reed computer science faculty for clarification on a problem's statement. You cannot, however, use an IDE, a compiler, the web, your notes, etc. to complete the exam, nor can you consult any other individuals for help. Doing so is a violation of the honor principle.

Note that you are welcome to write additional functions that support the code you are asked to write, should you feel the need to do so. (If the instructions ask you to "Write a function that..." that does not mean you cannot write some "helper" functions, as well.)

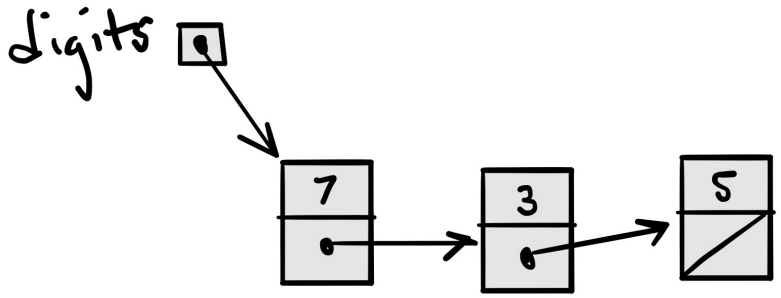
1. Consider the following C++ definition of a struct that serves as a linked list node for a sequence of digits:

```
struct DigitNode {
  int digit;
  struct DigitNode* next;
};
```

Consider the following class specification:

```
class DigitSequence {
private:
  DigitNode* digits;
public:
  DigitSequence(void);
  void increment(void);
  void decrement(void);
  ~DigitSequence(void);
};
```

It is meant to represent a sequence of digits of a non-negative integer as a linked list. The first node referenced by the pointer `digits` contains the least significant digit. The last node contains the most significant digit. The number 537 would be represented as a linked list with three nodes, the first containing 7, the second containing 3, and the last containing 5. Here is a picture of the representation of 537:



Note that the single-digit integers 0 through 9 have just one linked list node with that single digit.

Write each of the four methods according to the specs below.

- The first method `DigitSequence` is the constructor. It should build a linked list containing the single digit 0.
- The second method `increment` is a method that adds one to the number coded by the digit sequence. If that number happens to contain a sequence of 9 digits, for example 9999, then `increment` will have to change all the 9s to 0s and then allocate and link a new node containing the digit 1 to the end of the linked list.
- The third method `decrement` subtracts one from the number coded by the digit sequence, if the number is greater than 0. If it is 0, nothing is changed. If the digit sequence encodes a positive power of 10, for example 10000, then `decrement` will have to change all the 0s to 9s and remove the node at the end containing the digit 1. It should `delete` that node.
- The fourth method `~DigitSequence` is the destructor. It should give all the nodes that it has allocated back to the heap using `delete`.

2. Below is the code for a `Vehicle` class. Vehicles have a position (specified by x and y coordinates), a size of gas tank (in gallons), a current amount of gas, and a fuel efficiency (in miles per gallon). They move around using the `driveTo` method, but only if they have enough gas. The method returns `True` or `False` to indicate whether that drive was made. The tank is initially full of gas.

```
class Vehicle:
    def __init__(self, x, y, tankSize, mpg):
        self.setPosition(x, y)
        self.tankSize = tankSize
        self.gas = tankSize
        self.mpg = mpg
    def setPosition(self, x, y):
        self.xPosition = x
        self.yPosition = y
    def distanceTo(self,x,y):
        xp = self.xPosition
        yp = self.yPosition
        return ((xp-x)**2 + (yp-y)**2)**.5
    def driveTo(self, newX, newY):
        distance = self.distanceTo(newX, newY)
        if distance <= self.gas * self.mpg:
            self.gas -= distance / self.mpg
            self.setPosition(newX, newY)
            return True
        else:
            return False
```

Write the code for a new class, `TowTruck`. This is a vehicle that is either unloaded or is towing a vehicle. All tow trucks have 30 gallons of gas. They are initially unloaded, not towing a vehicle. When unloaded, their fuel efficiency is 15 miles per gallon (MPG). Tow trucks drive around like normal vehicles, but also pick up and drop off the vehicles they tow. The class has two additional methods, `pickUp` and `dropOff`.

The `pickUp` method takes a vehicle as a parameter. When invoked, the tow truck drives to that vehicle and picks it up, but only if it is unloaded and has enough gas to reach it. It should return `True` if the vehicle is picked up, and `False` otherwise.

Driving around using `driveTo`, the positions of the tow truck and its towed vehicle change, but only the tow truck uses gas with each segment of driving. The fuel efficiency of the tow truck is 10 MPG when it is towing a vehicle.

The `dropOff` method does not take any parameters and simply unloads the tow truck from the vehicle, decoupling their future movement.

Write this code compactly and cleanly using the object-oriented features of Python, including judicious use of inheritance.

3. The bottom of this page has a summary of the MIPS32 instruction set, but only for a limited number of its instructions.

   Assume we have the start of a MIPS program with a label `text` of a null-terminated string of characters in memory. **Using only the instructions listed in this guide, write a snippet of MIPS assembly code** that modifies the contents of this string so that it is "rotated" to the left by one character, moving the first character to the last position. If, for example, the string is made up of 6 bytes whose non-null characters read "hello", then after your snippet executes it should instead read "elloh".

MIPS32 coding guide:

`li` $RD, *value*	–loads an immediate value into a register.
`la` $RD, *label*	–loads the address of a label into a register.
`lb` $RD, ($RS)	–loads a byte from memory at an address specified in a register.
`lb` $RD, *offset* ($RS)	–same, but at a constant offset from the given address.
`sb` $RS, ($RD)	–stores a byte of a register into memory at an address specified in a register.
`sb` $RS, *offset* ($RD)	–same, but at a constant offset from the given address.
`add` $RD, $RS1, $RS2,	–add two registers, storing the sum in another.
`sub` $RD, $RS1, $RS2,	–subtract two registers, storing the difference in a third.
`addi` $RD, $RS, *value*	–add a value to a register, storing the sum in another.
`shl` $RD, $RS, *positions*	–shift a register's bits left, storing the result in another.
`move` $RD, $RS	–copy a register's value to another.
`b` *label*	–jump to a labelled line.
`blt` $RS1, $RS2, *label*	–jump to a labelled line if one register's value is less than another.
`bltz` $RS, *label*	–jump to a labelled line if a register's value is less than zero.
`gt`, `le`, `ge`, `eq`, `ne`	–other conditions than `lt`

The registers you can access are named `$v0-v1`, `$a0-a3`, `$t0-t9`, `$s0-s7`, `$sp`, `$fp`, and `$ra`.

4. Below we ask you to represent integer sequences using the C++ `std::vector` class from the standard template library. **A useful method to recall** on vectors is the method `push_back`.

   (a) **Write a function** `splitInto` that takes three arguments, each being a vector of integers. The first is named `array` and is passed by value. The second named `evens` is passed by reference. The third is named `odds` and is also passed by reference. You can assume that `array` has a size of at least two, and that `evens` and `odds` each have size 0. Upon completion of `splitInto`, `evens` should contain the sequence of items of `array` at even positions (item 0, item 2, item 4, etc.) and `odds` should contain the sequence of items of `array` at odd positions (item 1, item 3, item 5, etc.).

   (b) **Write a function** `merge` that takes two arguments, each being a vector of integers, named `array1` and `array2`, and passed by value. You can assume that `array1` and `array2` each have a size of at least one and are sorted. The function should return a new vector containing the items of both, and in sorted order. For example, if the first contains the sequence 0,3,4,5,18 and the second contains the sequence 2,4,6,7,11 then it should give back a new vector with the sequence 0,2,3,4,4,5,6,7,11,18.

   (c) Now **write a recursive function** `mergeSort` that uses these two functions. It takes an integer vector named `array` and it is passed by value. It should return back an integer vector with the same size and storing the same integer values, but in sorted order. It should do this using this algorithm:

   > *If the vector it is given has size less than two, then that array is already sorted, so it can just return that array back. If the size is two or greater, then it splits the vector into two vectors, the even-indexed items and the odd-indexed items, with* `splitInto`*. It then calls* `mergeSort` *on each of those vectors individually, thus making two different recursive calls. It gets back two sorted vectors, one with each call. It then merges these two vectors into a single vector using* `merge` *and returns that sorted result.*

   When given a vector containing 18,7,4,6,5,4,3,2,0,11, it should split them into 18,4,5,3,0 and 7,6,4,2,11. It then sorts each making two new vectors with 0,3,4,5,18 and 2,4,6,7,11. It then merges them into a new vector with 0,2,3,4,4,5,6,7,11,18.

5. For each of the Python functions below, **give the asymptotic runtime of the function.** Each takes a list `xs` as its argument. For each, you should give the runtime as a function of n, where n is the length of the list. You can assume that accessing an element of the list takes constant time, and that obtaining the length (with `len`) takes constant time.

(a)
```
def f(xs):
    # Sum up the values in the list.
    i = len(xs)
    s = 0
    while i > 0:
        s = s + xs[i]
        i = i - 1

    # Check if the sum is above 1000.
    if s > 1000:
        return True
    else:
        return False
```

(b)
```
def g(xs):
    i = 0
    while i < len(xs):
        j = 0
        while j < len(xs):
            # Do two list elements sum to 100?
            if xs[i]+xs[j] == 100:
                return True
            j = j+1
        i = i+1
    return False
```

(c)
```
def h(xs):
    y = f(xs)           # Execute algorithm f on the list
    z = g(xs)           # Execute algorithm g on the list
    return y and z
```

(d)
```
def one_last(xs):
    i = len(xs)
    a = 0
    while i > 0:
        a = a + xs[i]
        i = i // 2      # Keep shrinking i by dividing it by 2.
    return a
```

Computer Science Junior Qualifying Exam

Discrete Structures Session

10am-noon, August 27, 2020

Computer Science Department, Reed College

You have two hours to complete this portion of the exam. There are five problems on six pages. Handwrite your answers on blank sheets of paper, writing your name and the number of the problem you are answering clearly on each sheet.

Complete each problem to the best of your ability. When the problem asks you for a formal proof or a justification, please take care to communicate enough detail to us. For most problems, just giving the final answer is not enough. Some justification is required.

You may ask members of the Reed computer science faculty for clarification on a problem's statement. You cannot, however, consult textbooks, the web, your notes, etc. to complete the exam, nor any other individuals for help. Doing so is a violation of the honor principle.

1. You decide to host a Reed social to welcome the first year students interested in computability theory. 6 of these students are Philosophy majors, and the other 24 are CS majors. You've chosen to serve pizza at the event. You've ordered three 12-slice mushroom pizzas and five 12-slice cheese pizzas. In summary:

   - $\frac{1}{5}$ first year students attending this event major in Philosophy.
   - $\frac{4}{5}$ first year students attending this event major in CS.
   - $\frac{3}{8}$ pizza slices served at this event are topped with mushrooms.
   - $\frac{5}{8}$ pizza slices served at this event are plain cheese.

   It turns out that Reed first year students interested in computability theory choose their slice of pizza at random, but how they do this depends on their prospective major. If they plan to be a Philosophy major, when given the option of a mushroom slice and a cheese slice, they choose one of the two toppings at random, regardless of the number of slices available for each topping, and grab a slice with that topping. If instead they plan to be a CS major, they pick a slice at random from the slices that are available, with no concern about the toppings.

   The event is just about to begin and a first-year, arriving early, walks into the room. They grab a slice of mushroom pizza. **What is the probability** that they are planning to be a Philosophy major?

2. Here we consider Fermat's Theorem (sometimes called his "little theorem") that states that, for p a prime integer, and for any integer a that is relatively prime to p, we have that

$$p \mid a^p - a \tag{1}$$

that is, p wholly divides $a^p - a$. This is sometimes equivalently stated as

$$a^{p-1} \equiv 1 \pmod{p} \tag{2}$$

(a) **Give a formal proof** that the first relation (1) implies the second relation (2).

(b) **For what number** $x \in \{0, 1, 2, \ldots, 112\}$ is the following true?

$$11^{112114} \equiv x \pmod{113}$$

3. Consider the following two Python functions:

```
def f(n):
  if n < 3:
    return 1
  else:
    return f(n-1) + 2 * f(n-2) + f(n-3)

def g(n):
  return f(n) % 3
```

Give a formal proof that `g` returns 1 when fed any integer input.

4. Consider rolling two fair, 6-sided dice, and then summing the result. For example, you might roll a 3 and a 5, and their sum would be 8.

   (a) **What is the probability** that the rolled dice sum to 4? **What is the probability** that the rolled dice sum is 7?

   (b) **What is the expected value** of a rolled sum?

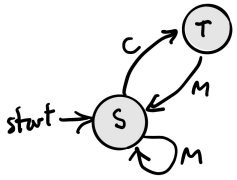
   (c) Suppose you roll the two dice over and over until the dice sum to 4 or sum to 7. **What is the number** of rolls you'd **expect** to make (counting that last roll of 4 or 7)?

5. The night before major exams, Alice and Bob each like to study with their favorite comfort food, cookies and milk. They snack slightly differently, though. In this problem, we count the number of ways they might each pattern their snacking while studying.

   (a) Alice always takes a sip of milk after each cookie she eats. And then sometimes she takes a sip of milk on its own. For example, she might start by eating a cookie. She then follows it with a sip of milk. She might repeat that, one cookie, one sip. And then maybe she takes a third sip of milk.

   We'd say her snacking pattern, then, was CMCMM. Two cookies and three sips of milk, but interleaved in that way. Another pattern of Alice's could be MMMCM-MMCM, but notice there is always an sip M after a cookie C.

   We've analyzed Alice's behavior and have this diagram of Alice's snacking process:

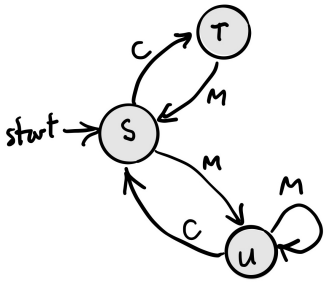


   Alice starts studying in state S. In state T she's just had a cookie, but still needs to take a sip of milk to wash it down in order to return to state S. So Alice will never stop snacking in state T. She can also just take a sip of milk and remain in state S. She can stop snacking in state S.

   Last night, in prepping for today's qual, Alice ate 3 cookies and took 8 sips of milk. **In how many different** patterns could Alice have eaten her snack?

   (b) Bob also pairs a sip of milk with each cookie he eats, sometimes sipping the milk in preparation for eating a single cookie, other times washing down a cookie he just ate with a sip of milk. An example of one of these patterns of Bob is CMMMMCCM. Here he eats a cookie then washes it down with a sip of milk, he then takes two sips of milk, then sips milk and eats a cookie, then has the last cookie followed by that last sip of milk. All in all, he ate three cookies and took five sips of milk in this pattern.

   We've analyzed Bob's behavior and have this diagram of Bob's snacking process:



   Bob starts studying in state S. In state T he's just had a cookie, but still needs to take a sip of milk to wash it down in order to return to state S. So Bob will never stop snacking in state T. When in state U, Bob's just had a sip of milk, and so he can then eat a cookie putting him back into state S. Or he can sip more milk, staying in state U. He can stop snacking either in state S or in state U.

   We learn that last night Bob had 3 cookies along with 7 sips of milk. **How many different** snack patterns are possible, given the way Bob snacks?